Roslyn

Succinctly

by Alessandro Del Sole

Syncfusion | Technology Resource Portal

Roslyn Succinctly

By
Alessandro Del Sole

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Alessandro Del Sole has been a Microsoft Most Valuable Professional (MVP) since 2008. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally considered a Visual Studio expert and a Visual Basic authority.

Alessandro has authored many printed books and e-books on programming with Visual Studio, including *Visual Studio 2015 Succinctly*, *Visual Basic 2015 Unleashed*, *Visual Studio LightSwitch Unleashed*, and *Visual Basic 2012 Unleashed*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals, including the Visual Basic Developer Center from Microsoft. He is a frequent speaker at Italian conferences, and has released a number of Windows Store apps. He has also produced a number of instructional videos in both English and Italian. You can follow him on Twitter at @progalex. Recently, Alessandro co-founded the Roslyn Academy, an online community that shares articles, videos, code, and experiences about the .NET Compiler Platform.

Introduction

In the last couple of years, Microsoft has been embracing the open source world in many ways, changing their historic philosophy. Embracing open source means two important things. The first is that Microsoft is now also building services and applications for open source systems. The second is that Microsoft is releasing many applications, tools, and services as open source projects, which is a revolution. Many building blocks in the .NET Framework architecture are now open source projects. Among others, Microsoft has rewritten the Visual Basic and C# compilers entirely in managed code, releasing them as an open source project called the .NET Compiler Platform, also known as Project Roslyn. The .NET Compiler Platform is new stuff for developers who build tools for other developers. If you build and sell libraries, extensions, user controls, or even stand-alone development tools, the .NET Compiler Platform is definitely for you.

This book has many goals. The first goal is to introduce the .NET Compiler Platform, with a high-level overview of the platform. The second goal is to explain why it's so important and what you can do with it. The third goal is to show how to use the .NET Compiler Platform in practice, both with tools already included in Visual Studio 2015, and with tools you create. The last goal is to show how to share the goodies you create with the world. However, this book is not for beginners. There are many important concepts about .NET development that you must already know, such as what a .NET assembly is and how one is composed and compiled, what IL code is, the async/await programming pattern, reflection, and of course, concepts related to object-oriented programming.

In terms of software requirements, you need Visual Studio Community 2015 (VS 14.0) or higher. Express editions are not supported. You also need the Extensibility Tools for Visual Studio, which you can get by installing the Visual Studio 2015 SDK.

The full source code for all of the examples in this book is available on GitHub.

After reading this book, I'm sure you will be excited about the new opportunities Roslyn offers to create amazing developer tools and improved coding experiences for your customers.

This book is dedicated to my dad. You're a great man and you deserve the best for the rest of your life.

Alessandro

Chapter 1 Project Roslyn: The .NET Compiler Platform

With the recent release of Visual Studio 2015 and .NET Framework 4.6, Microsoft has also shipped version 1.0 of the .NET Compiler Platform, formerly Project Roslyn, which provides open source C# and Visual Basic compilers with rich code analysis APIs. The .NET Compiler Platform plays a crucial role in the new .NET ecosystem and empowers Visual Studio 2015 in many ways. For a better understanding of what the .NET Compiler Platform is and why it's so important, you need some explanation about what's new with compilers in the latest release. This chapter provides an introduction to Roslyn, giving you the foundation to understand the more complex concepts described throughout the book that are necessary before you start writing code.

 ***Note: Except for where expressly stated, all the topics described in this book apply to both Visual Basic and C#.***

Introducing Project Roslyn

Over the last few years, Microsoft has been very busy rewriting the C# and Visual Basic compilers entirely in managed, .NET code. Other than being rewritten, both compilers expose rich code analysis APIs you can leverage to build developer tools that perform code-related tasks, and are the same ones used by the Visual Studio code editor to detect issues and errors as you type. By exposing APIs, compilers are treated as a platform (also known as compiler-as-a-service); this opens an infinite number of possible development scenarios with particular regard to code analysis and code generation.

Additionally, Microsoft has released the new Visual Basic and C# compilers as an open source project, which means that you can download the source code, investigate it, and collaborate by sending your code, feedback, or suggestions. These rewritten, open source compilers exposing rich code analysis APIs use the name .NET Compiler Platform.

 ***Note: This book uses the ".NET Compiler Platform" and "Roslyn" names interchangeably. The first name is the product's name at release time, but the Roslyn alias has been (and is still) widely used for years to refer to the project. In fact, many developers around the world still prefer using the Roslyn alias rather than the official name.***

As for all the other open source projects from Microsoft, the .NET Compiler Platform is hosted on GitHub, a very popular website that allows sharing code and offers tools for collaboration, including source control based on the Git engine, and tools for accepting contributions from the developer community. The official .NET Compiler Platform home page on GitHub contains the full source code for compilers and the APIs they expose, the documentation, the API reference, and code examples. It is worth mentioning that the project home page also contains

documentation for new language features in C# 6.0 and Visual Basic 14. The implementation of some of the new language features has been possible thanks to the .NET Compiler Platform.

How Does Roslyn Change the Way I Write Code?

The Visual Basic and C# compilers have been rewritten in managed code and now expose APIs, but from a practical point of view, this does not really change the way you use programming languages and write code. However, behind the scenes, the code editor in Visual Studio 2015 strongly relies on the .NET Compiler Platform. For example, live code analysis that detects issues as you type, which you already know from the past, is now built upon the Roslyn APIs. Also, Visual Basic now supports code refactorings and C# has an improved refactoring experience; both have been rebuilt on top of the .NET Compiler Platform. I'll describe these improvements in detail in Chapter 2, so do not be afraid if something is not clear at this point.

Roslyn in Practice

You use the Roslyn APIs to build tools for developers. More specifically, with the Roslyn APIs you can:

- Write and share custom, domain-specific code analysis rules that integrate into the Visual Studio 2015 code editor. In this way, you can extend the live code analysis engine with your own rules. This means that you can write diagnostics and code fixes (known as **analyzers**), and code refactorings for your APIs or for specific programming patterns, and the Visual Studio code editor will detect code issues as you type, squiggling the code that needs attention, and suggesting the proper fixes. Writing diagnostics and code fixes is discussed in Chapter 4, and writing code refactorings is discussed in Chapter 5.
- Implement code generation, emit IL code, and perform other code-related tasks inside your .NET applications with the compiler APIs.
- Build stand-alone tools that run outside Visual Studio, but can leverage MSBuild workspaces and take advantage of the Visual Basic and C# compilers to perform code-related tasks.
- Create an interactive, interpreted C# or Visual Basic programming environment using a read-evaluate-print loop (REPL).

The possibilities are not limited to this list. Wherever you might need to analyze source code or build tools that leverage the compiler's APIs, that's where the .NET Compiler Platform comes in.

Cloning the Roslyn Repository to Your Computer

To experiment with Roslyn, you will need to download and install the .NET Compiler Platform SDK, as is explained in the next section. But because Roslyn is an open source project on GitHub, you can also optionally view and modify Roslyn's source code, as is explained in this section.

Tip: You need a GitHub account to complete the steps described in this section. If you do not have one, you can register for free. Also, the GitHub extension for Visual Studio

2015 must be installed. If it is not installed, you can update Visual Studio's installed components from the Windows Programs and Features tool.

GitHub and the Git source control engine allow cloning a source code repository on local machines, which is very useful for editing and testing any open source project locally, and for familiarizing yourself with the code. Of course, this is the case with the Roslyn project, too. To clone the full Roslyn source code, which includes the compilers' source code, follow these steps:

1. Open your favorite web browser, navigate to https://github.com/dotnet/roslyn, and sign in with your account (see Figure 1).
2. Click **Open in Visual Studio**.
3. When Visual Studio 2015 starts, the Team Explorer window automatically appears, supplying the necessary information to clone the project (see Figure 2). You can provide a different local folder if you want. When ready, click **Clone**. If you've never signed into GitHub from Team Explorer, Visual Studio will ask you to enter your GitHub credentials.
4. When the operation completes, Team Explorer shows the full list of Visual Studio solutions in the code repository, plus a number of shortcuts that you can use to manage the project (see Figure 3).
5. Double-click the **Roslyn.sln** file and wait for Visual Studio 2015 to open all of the Roslyn project's code.

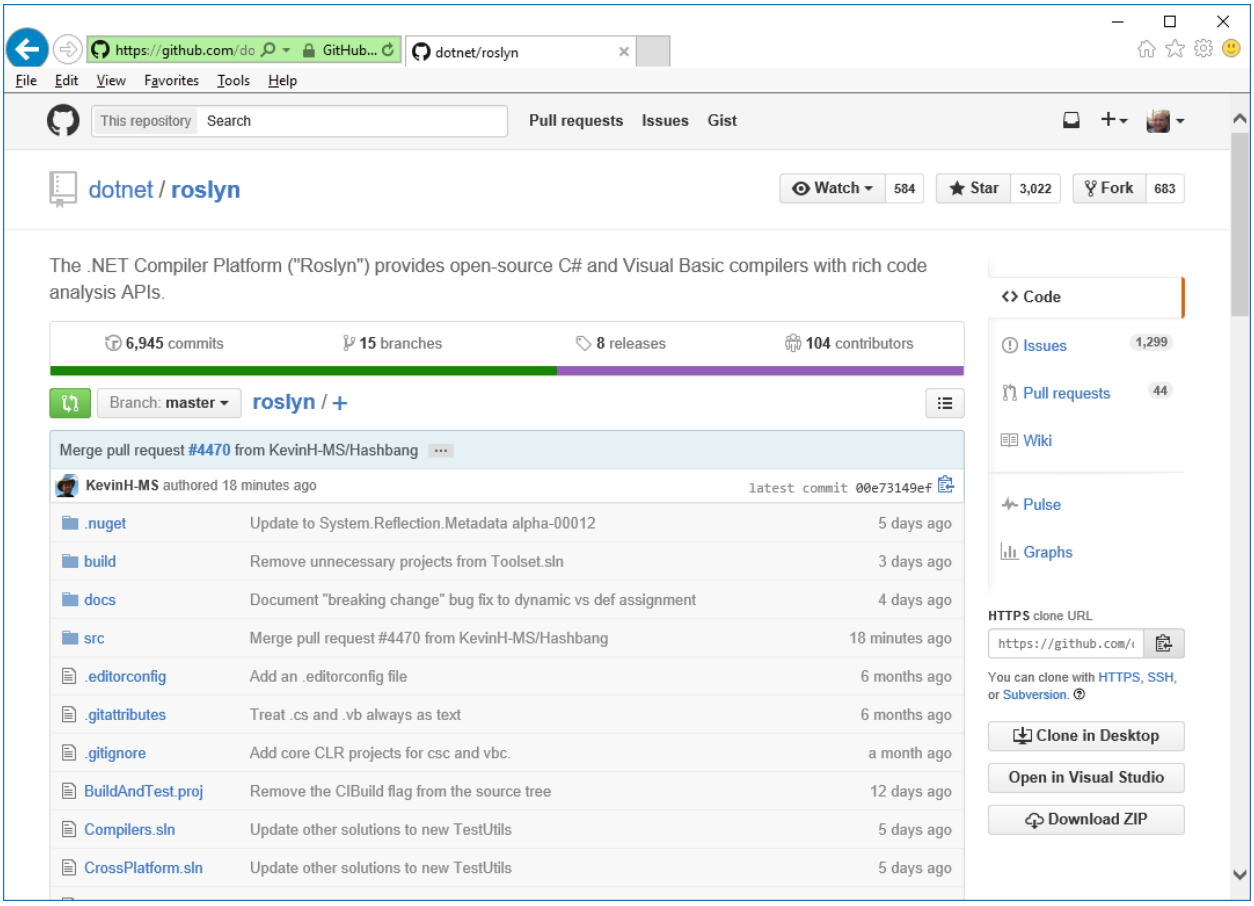

Figure 1: The Roslyn home page on GitHub

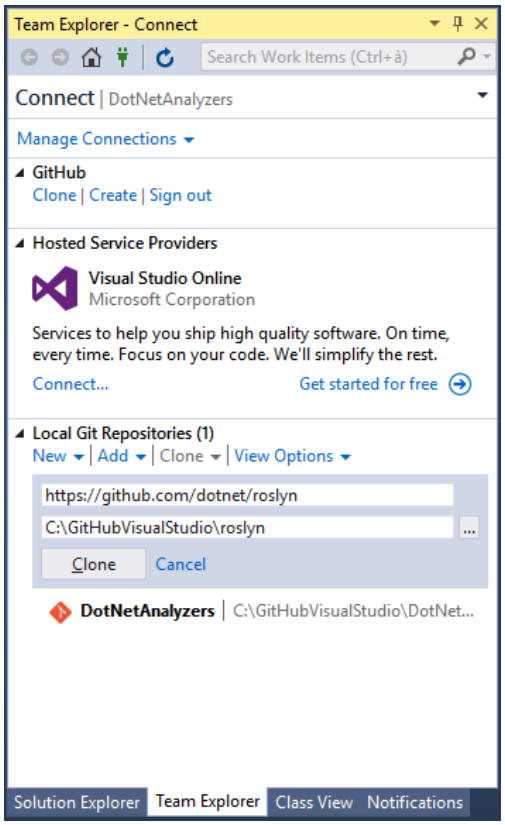

Figure 2: The Team Explorer window ready to connect to GitHub

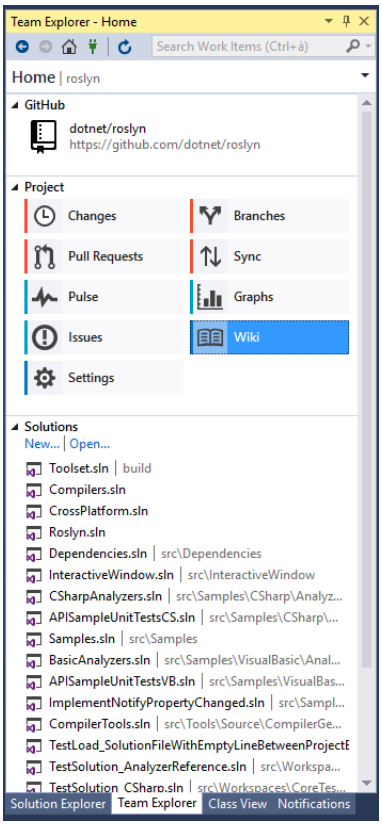

Figure 3: The Roslyn project's source code cloned locally

You can now explore the source code of the entire .NET Compiler Platform, including compilers, features such as Visual Studio's tool windows, refactoring tools, and built-in code analysis rules. You can also make changes to the code (do it only if you have experience with compilers) and test your edits locally. To do this, set the **OpenSourceDebug** project as the startup project and then press **F5**. This will launch the so-called Roslyn Hive, an experimental instance of Visual Studio where you can debug the source code of the .NET Compiler Platform. You can also compile and run unit tests included in the solution. To do so, write the following line in the Visual Studio Command Prompt:

```
msbuild /v:m /m BuildAndTest.proj /p:PublicBuild=true
/p:DeployExtension=false
```

This command will build and run the unit tests supported in the current public build of Visual Studio. Additional information about debugging, testing, and running the source code can be found on the Building, Testing, and Debugging page of the Roslyn project.

Collaboration: Sending Feedback, Suggestions, and Requests

Microsoft has been publishing all of its open source projects to GitHub. This website provides a full collaboration platform, which not only includes source control, but also specific tools to

submit your feedback, suggestions, and requests. This allows direct engagement between the team and the developer community. For instance, you might want to submit a bug, request a feature, or even send a piece of code you added to the cloned Roslyn project that you think should be evaluated for future releases. In order to send your contribution, you must respect the Contributing Code guidelines. All of your submissions will be deeply evaluated according to the terms of the guidelines. If you want to send feedback, you can open an issue or send a pull request (do not forget to search for similar issues or requests before you submit a new one). This book is not intended to be a full guide to the GitHub tooling, so make sure you read the Help page if you want to actively collaborate on the Roslyn project.

Installing the .NET Compiler Platform SDK

In order to write code analyzers, refactorings, and stand-alone .NET applications that leverage the Roslyn APIs, you first need to install the .NET Compiler Platform SDK, which you will need to complete the next chapters. The SDK is a free extension for Visual Studio 2015 that can also be installed from within the IDE. To do this, open Visual Studio 2015 and select **Tools** > **Extensions and Updates**. In the Extensions and Updates dialog box, select **Online** on the left, and then in the search box, type **.NET Compiler Platform SDK** (see Figure 4).

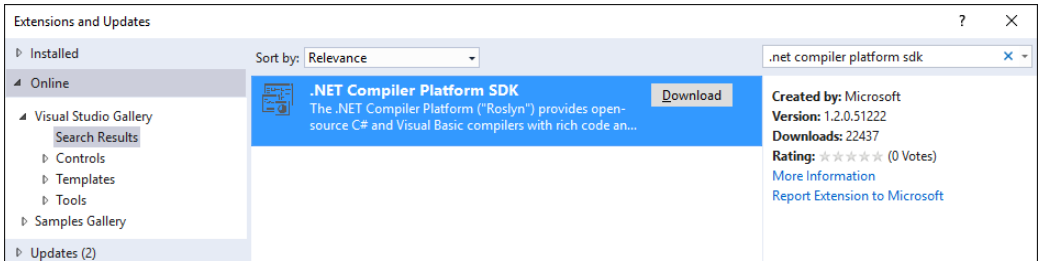

Figure 4: Installing the .NET Compiler Platform SDK

On your machine, click **Download**. After restarting Visual Studio, you are ready to go. The .NET Compiler Platform SDK includes project and item templates required to create diagnostics, code fixes, code refactorings, and stand-alone, Roslyn-based .NET applications. It also includes a very important tool called Syntax Visualizer that is discussed later in Chapter 3, and will be used frequently throughout this book.

What's Next

Leveraging the Roslyn APIs to build code analysis tools is not difficult once you understand the purpose of the .NET Compiler Platform (and, of course, if you have some experience with .NET and Visual Studio). In order to help you get the best understanding possible of the .NET Compiler Platform and what you can do with it, topics in this book have been organized in the following sequence:

1. Explaining and showing how Visual Studio 2015 uses the .NET Compiler Platform to enable the live code analysis engine in the code editor to detect code issues as you type (see Chapter 2).

2. Describing the Roslyn APIs and architecture with technical explanations and diagrams, and the concept of **syntax**, including explanations on how to investigate the syntactic structure of existing code with the help of the Syntax Visualizer tool (see Chapter 3).
3. Guiding you to the most relevant topics of the book: writing custom, domain-specific code analyzers and refactorings (see Chapters 4 and 5).
4. Helping you share your tools with other developers, either via NuGet or the Visual Studio Gallery (see Chapters 6 and 7).
5. Giving you the basics to use APIs for compiling code and managing solutions and projects (see Chapter 8).

If you are like me, you might be impatient to jump to Chapter 4 and put your hands on the keyboard. However, it is strongly recommended that you read all chapters in sequence. After all, they are not very long, and I promise they will not be annoying.

Chapter Summary

The .NET Compiler Platform, formerly Project Roslyn, provides open source Visual Basic and C# compilers with rich code analysis APIs. This chapter provided an introduction to the .NET Compiler Platform, describing how the Visual Basic and C# compilers have been rewritten in managed code and how they expose APIs that developers can leverage to build developer tools, such as diagnostics, code fixes, and code refactorings that integrate into the Visual Studio 2015 code editor or stand-alone code analysis tools. You saw how the full source code for the .NET Compiler Platform is available on GitHub, how you can clone the code repository onto your computer for local tests, and what tools you need in order to work with the .NET Compiler Platform. The next chapter describes how Visual Studio 2015 uses Roslyn for an improved coding experience; this will give you a clearer idea of what you can do with Roslyn's analysis APIs.

Chapter 2 Coding in Visual Studio 2015: A Roslyn-Powered Experience

The best way to get a more precise idea of what the Roslyn APIs can be used for is to look at how Microsoft used the new compilers' services in Visual Studio 2015. The code editor has been enhanced and brings significant improvements to the coding experience. Some existing tools have been improved, and new features have been added to help developers be more productive than ever by keeping their focus on the active editor window while fixing issues and refactoring code. This improved experience is also known as **code-focused experience**, and is completely powered by the .NET Compiler Platform. This chapter describes updated and new features in the code editor. In this way, you will become more familiar with Roslyn, starting from how Visual Studio 2015 itself uses the platform.

Live Static Code Analysis: Light Bulbs and Quick Actions

Though restricted to specific code issues, the Visual Studio code editor has always offered live static code analysis, which means that background compilers could detect code issues and report warnings and errors by squiggling the code that needs your attention, and populating the Error List window with detailed error or warning messages. Figure 5 shows an example of live static code analysis with code that declares a variable of type **`int`** which is never used, and attempts to call a **`DoSomething`** method that does not exist.

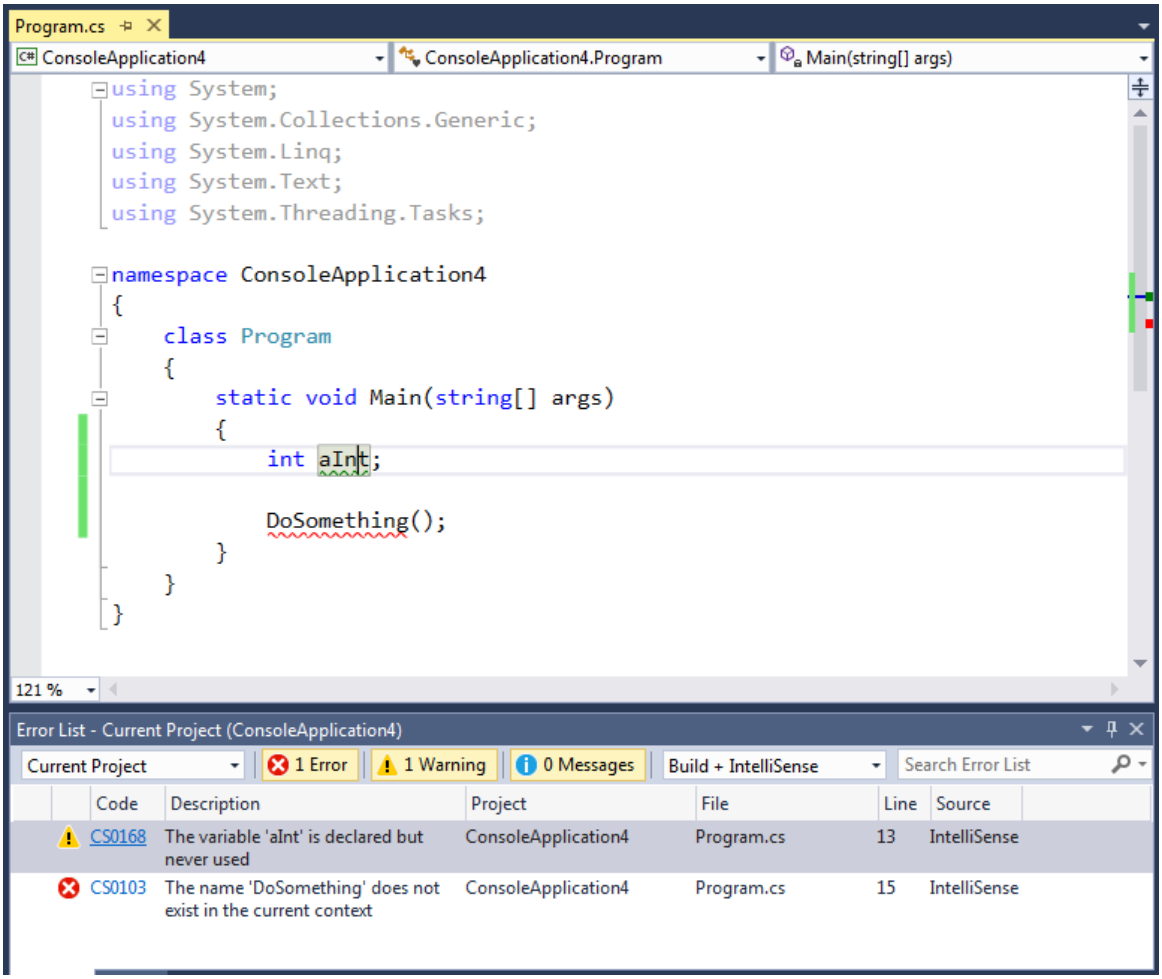

Figure 5: Live code analysis in Visual Studio

However, the code analysis engine prior to Visual Studio 2015 had the following limitations:

- Most code issues could be detected at compile-time only.
- The code editor could only analyze code for a number of rules coded at Microsoft, with no way of adding custom rules.

In Visual Studio 2015, the live, static code analysis engine has been completely rewritten on top of the .NET Compiler Platform. This changes and improves the way you can fix code issues and allows for creating and integrating custom analysis rules, as will be explained in Chapters 4 and 5. For instance, if you hover over an error in the code editor with your pointer, an icon called the **light bulb** appears (see Figure 6).

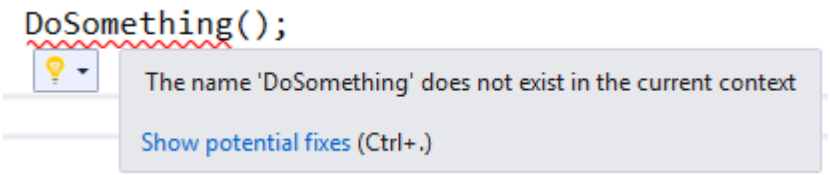

Figure 6: The light bulb over an invalid line of code

If you either click the light bulb or the **Show potential fixes** hyperlink, Visual Studio shows a number of potential fixes (known as **quick actions**), as demonstrated in Figure 7.

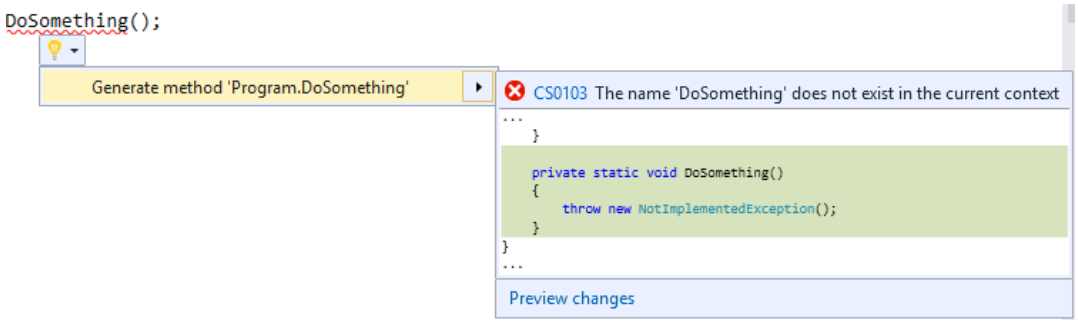

Figure 7: Quick actions offered to fix the current issue

For each quick action, Visual Studio shows a live preview that highlights lines of code that will be added in green, and highlights lines of code that will be removed in red. To apply the suggested fix, click the desired quick action. In this case, you would click **Generate method 'Program.DoSomething'**.

Tip: You can also enable light bulbs manually by pressing Ctrl+Period or by right-clicking in the code editor and then selecting** Quick Actions **from the pop-up menu. This is what you will do to apply code refactorings. Also, when you click a line of code containing an error, the light bulb automatically appears at the beginning of the line.

You can also get a more detailed preview by clicking **Preview changes**. This will launch the Preview Changes dialog box, which shows a preview of code changes and the list of files affected by the quick action. Figure 8 shows an example based on the current issue.

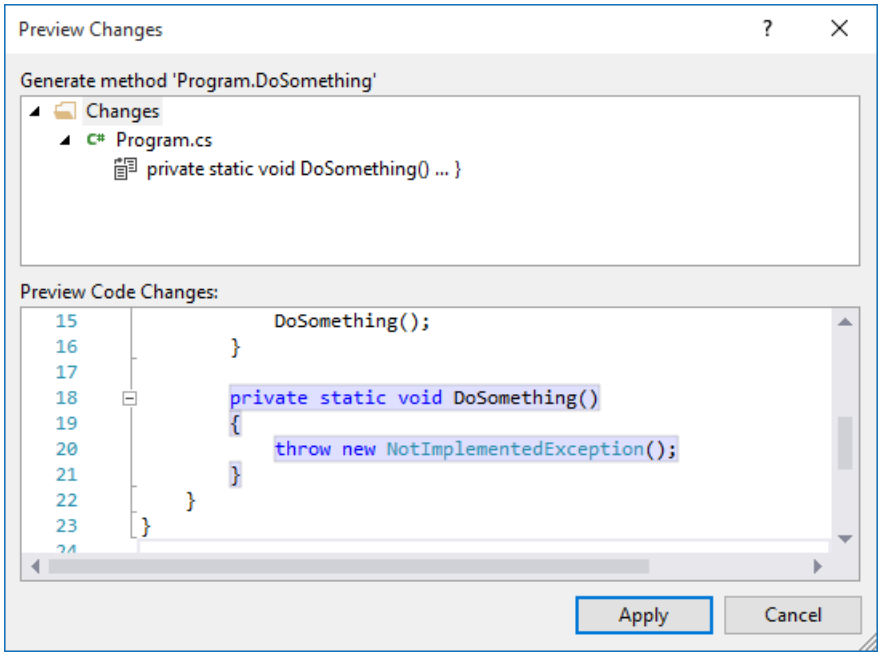

Figure 8: Previewing a quick action's changes

You can click **Apply** to apply the suggested fix or **Cancel** to ignore it. It is worth noting that in Visual Studio 2015, the Error List window has been completely rewritten to support the new code-focused experience. Among other changes, you will see that the Code column reports the error code in the form of a hyperlink; if you click it, Visual Studio 2015 will open a webpage containing information about the error (or a Bing search if no specific page is provided).

The previous example is a simple one, but the Visual Studio 2015 analysis engine is much more powerful. For example, suppose you have a class that must implement the **`IDisposable`** interface. If you declare that the class is going to implement the interface, but you do not supply the required code, not only is an error reported as you would expect, but a light bulb also suggests the appropriate fixes. Figure 9 shows how you can implement the interface with four different alternatives, and for each possible fix you can see the preview without losing focus on your code.

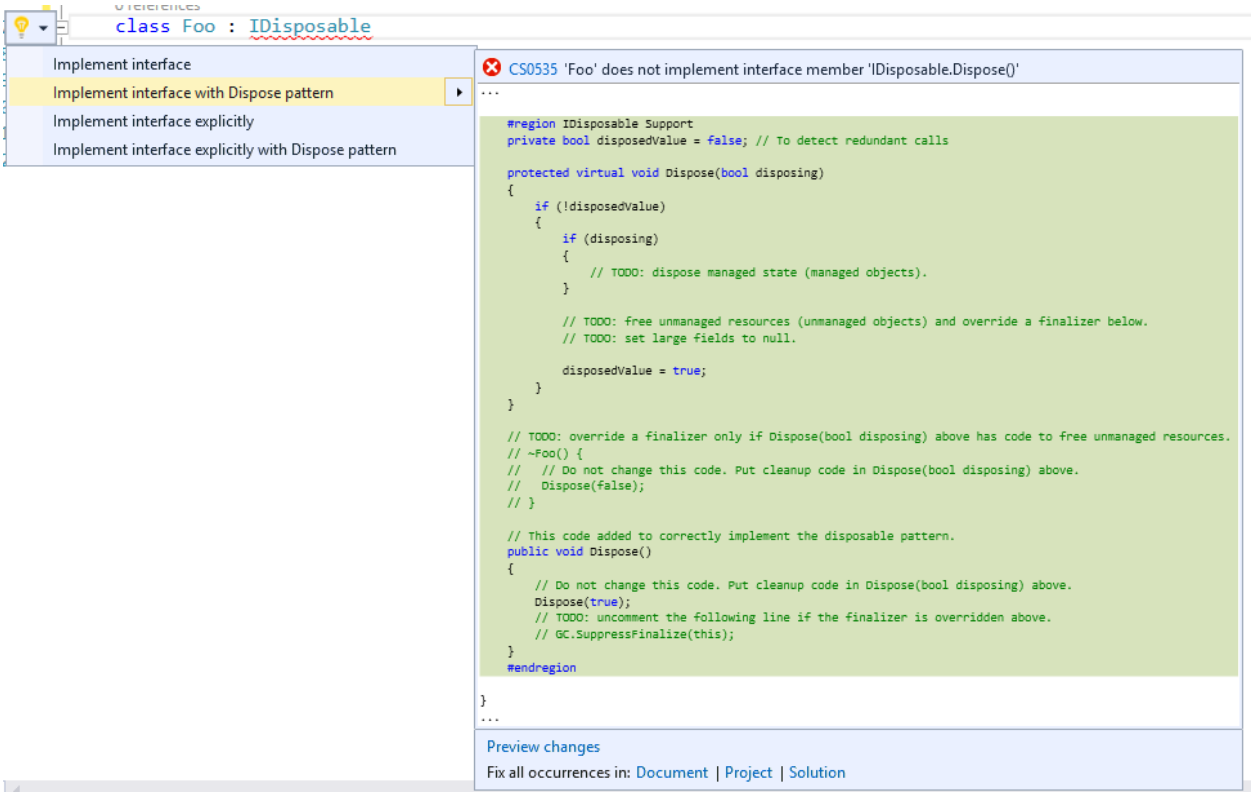

Figure 9: Visual Studio 2015 suggests contextualized code fixes

This is a tremendous benefit. Visual Studio 2015 not only saves you the time of writing the required code for a particular scenario, but it also shows the proper fixes for the context you are working on. Exploring the additional potential fixes for the **`IDisposable`** interface is left to you as an exercise. Of course, the **`IDisposable`** interface implementation is not the only scenario where Visual Studio can suggest multiple and contextualized fixes, but this is definitely a very common situation, and possibly the most effective one to make you understand the power of the new code analysis engine.

Code Refactoring

Refactoring is a technique that allows reorganizing portions of code a better way, without changing the original behavior. Refactoring is very common (and recommended) after writing many lines and blocks, because it improves code readability and maintainability. Visual Studio's code editor has always offered built-in support for refactoring in C#, but no support at all for Visual Basic (VB). In Visual Studio 2015, refactoring in C# has been improved and support for VB has been introduced for the first time, with tooling parity between languages. In Visual Studio 2015, refactoring tools are available through light bulbs and quick actions.

To demonstrate how this works in the new IDE, consider the example in Code Listing 1, which calculates the area of a circle, given the radius.

Code Listing 1: Calculating the area of a circle

```
static void Main(string[] args)
{
    double radius = 4;
    double circleArea = radius * radius * Math.PI;
    Console.WriteLine(circleArea);
    Console.ReadLine();
}
```

Select the whole method body and then enable the light bulb by right-clicking the selection and choosing **Quick Actions** from the pop-up menu. As you can see in Figure 10, the light bulb offers the **Extract Method** tool, whose purpose is encapsulating the selected code into a separate method for better readability and maintainability.

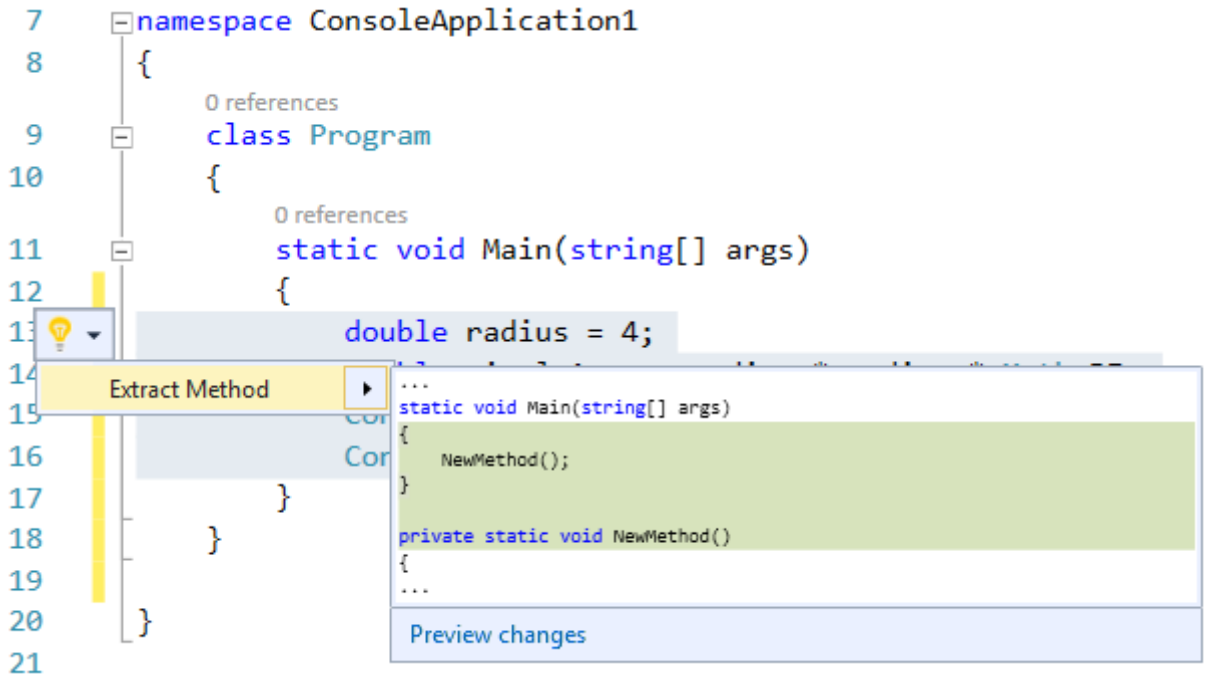

Figure 10: Code refactoring with the Extract Method tool

If you accept the suggested change, Visual Studio will extract a new method and ask you to supply a new name via the **inline rename** tool, which will replace all the occurrences of the previous name as you type (see Figure 11).

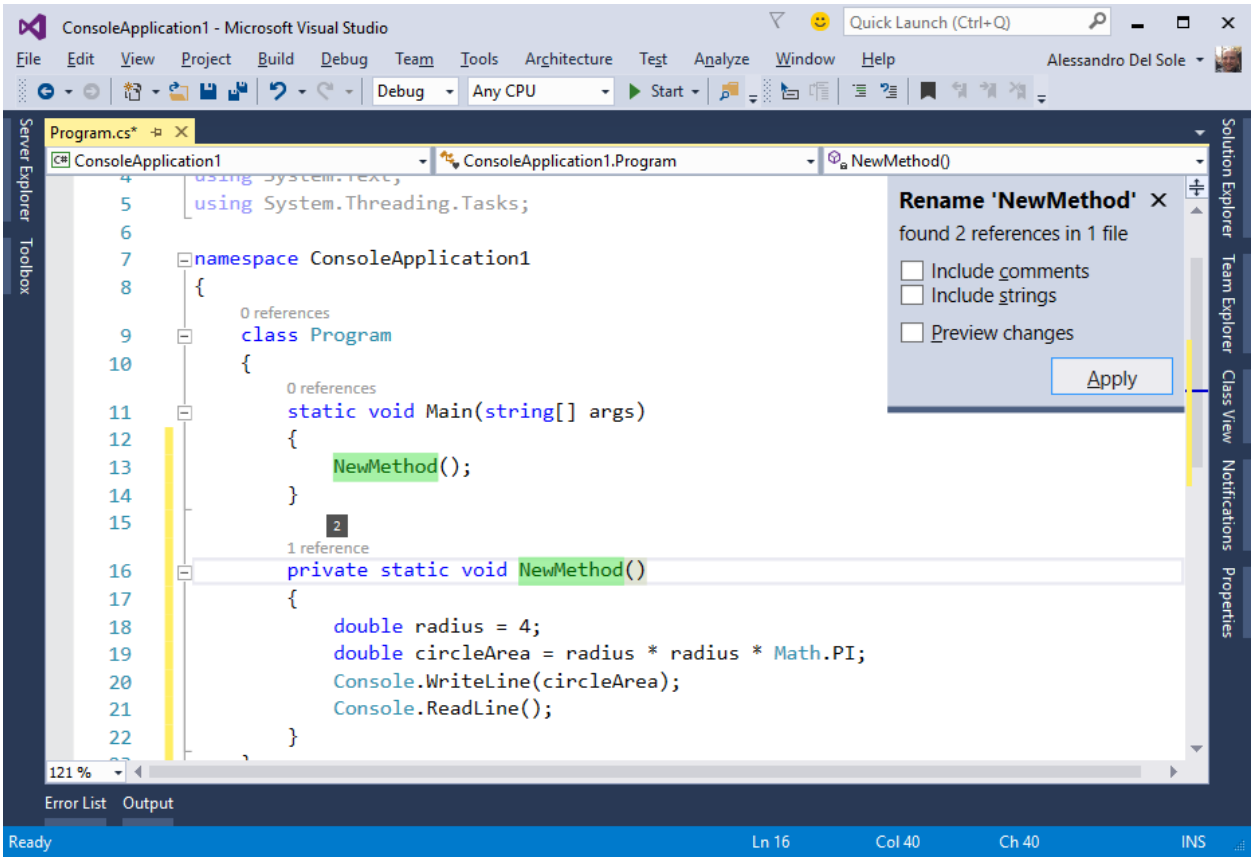

Figure 11: Extracting and renaming a method

Inline renaming is very straightforward because it also allows you to rename occurrences of an identifier inside string literals and comments, and replaces the old modal dialog used in previous versions of the IDE. With this new approach, developers can refactor code without losing focus on the editor, which always stays active. For a comprehensive list of available refactorings in Visual Studio 2015 for both C# and VB, you can download my free e-book, *Visual Studio 2015 Succinctly*. As with code analysis, you can get a more detailed preview before applying changes via the Preview Changes dialog box.

Removing Redundant Code

The code editor can automatically detect redundant code, such as unnecessary **`using`** (C#) and **`Imports`** (VB) directives, unnecessary **`this`** and **`Me`** keywords, or unnecessary conversions between types. Redundant code appears with lighter colors, such as light blue for reserved words, or light gray for identifiers. With light bulbs, you can easily refactor redundant code as demonstrated in Figure 12, which shows how to remove unnecessary **`using`** directives.

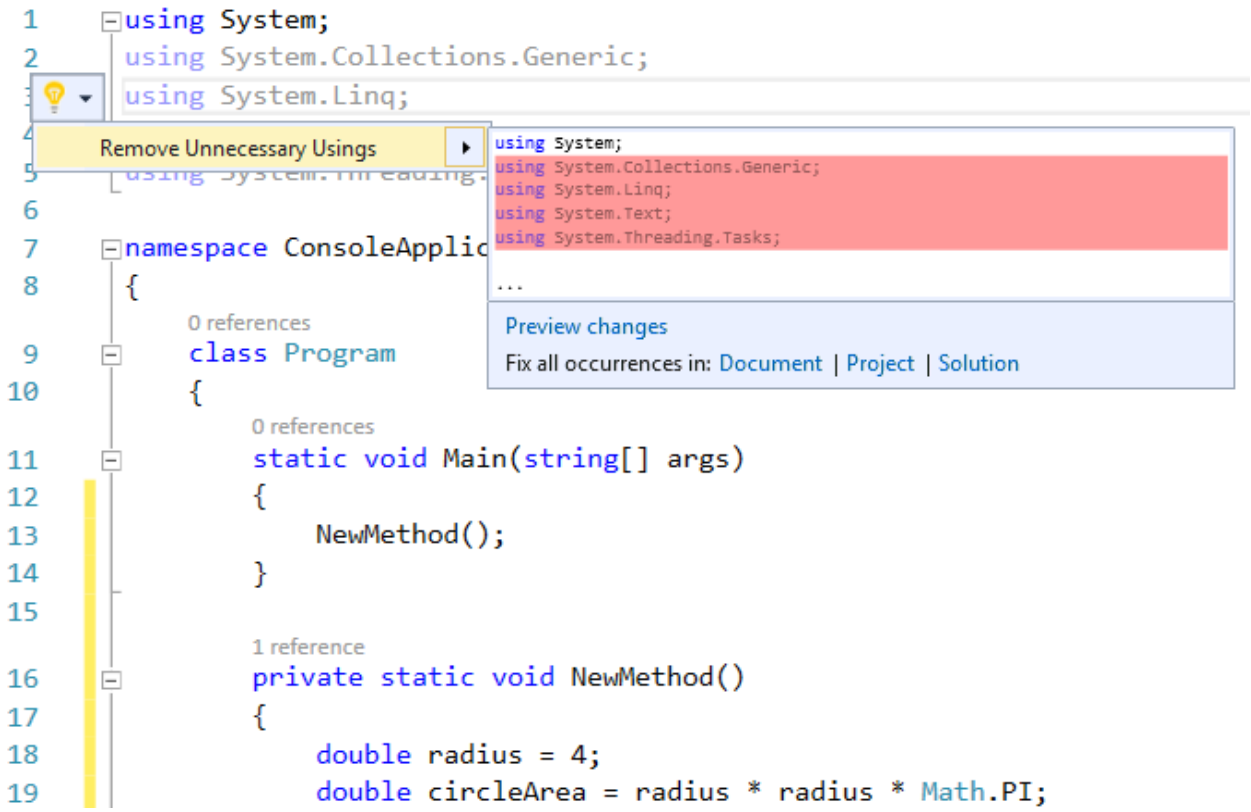

Figure 12: Removing unnecessary directives

As you can see in the live preview, the lines of code that will be removed are highlighted in red. Regarding **`using`** and **`Imports`** directives, if you select a number of directives and then right-click, the pop-up menu will show an item called **Organize Usings** for C#, or **Organize Imports** for VB, which exposes commands to remove directives, sort directives, or both.

Behind the Scenes: Roslyn Analysis Explained Succinctly

Behind the scenes, all the features described in this chapter so far are powered by the .NET Compiler Platform. To provide the coding experience described previously, the code editor uses a number of special plug-ins called **code analyzers** to analyze code in real time and detect code issues as you type. A code analyzer, or simply analyzer, is a library that uses the Roslyn APIs to analyze source code for syntax correctness, accuracy with the implementation of programming patterns, adherence to Microsoft's .NET coding rules, and so on.

Analyzers contain domain-specific rules and seamlessly integrate with the code editor. For each rule, an analyzer provides a **diagnostic**, which is responsible for code analysis on the single rule. Your code is automatically parsed at every key stroke; when a diagnostic detects code that is not compliant with a rule defined in the analyzer, it sends a notification and exposes possible fixes so that Visual Studio can report the specified error or warning, and provide the proper quick actions.

Visual Studio 2015 ships with a built-in set of analyzers that check for Microsoft's coding and syntax rules. This set is automatically added to every new project, but you can install additional analyzers and even create your own, as you will learn in Chapter 4, "Writing Code Analyzers". Similar concepts apply to **code refactorings**. Visual Studio 2015 ships with a built-in set of code refactorings that help you reorganize your code with common scenarios.

As for diagnostics, a code refactoring is exposed by a library the uses the Roslyn APIs and analyzes the source code to detect if any code blocks can be reorganized a better way. You can also install additional code refactorings, and even create custom ones, as you will learn in Chapter 5, "Writing Refactorings". Chapter 5 is a summary of how Roslyn empowers the code-focused experience in Visual Studio 2015, but starting with the next chapter, you will learn many more technical concepts required to get the most out of the .NET Compiler Platform.

Note: The code editor is not the only part of Visual Studio 2015 that relies on the .NET Compiler Platform. Language services, expression evaluators, and many tool windows and dialogs are also powered by the Roslyn APIs. To get an idea of what tools in Visual Studio 2015 are built on top of the .NET Compiler Platform, you can browse the source code online and take a look at folders whose names contain the word "Feature," plus the VisualStudio folder.

Downloading, Installing, and Using Code Analyzers

With Roslyn, developers can create custom analyzers and refactorings and publish them to the online NuGet package repository or the Visual Studio Gallery. Before you learn how to create and publish analyzers to both repositories, which is covered in the next chapters, it's important for you to know how to use existing analyzers by downloading, installing, and configuring these components inside a project. Because other developers will perform the same steps with your analyzers, you need to understand how they behave.

In the next example, you will download a sample analyzer I created that works with Windows Store apps, Windows Phone apps, and projects that use the OData client libraries from Microsoft. More specifically, this analyzer detects the usage of the **`System.DateTime`** type, and suggests using **`System.DateTimeOffset`** instead.

Tip: Controls like the DatePicker in Windows Store and Windows Phone apps work with objects of type `System.DateTimeOffset`, but passing an object of type `System.DateTime` is not reported as an issue. Also, the OData protocol works with `DateTimeOffset`, but passing `DateTime` is not reported as an issue. The sample analyzer helps passing the proper object type.

In Visual Studio 2015, create a new Windows Phone 8.1 project with the language of your choice (this example is in Visual Basic). Depending on which version of Visual Studio you are using, the Windows Phone application templates may not be installed by default. If you wish to try this example, you will have to download and install the Windows Phone SDK. You can do this from within Visual Studio in the **New Project** dialog after selecting the **Windows 8** project templates item. After installing the Windows Phone SDK, you can use the **Blank App (Windows Phone)** template, which is located under **Windows** > **Windows 8** > **Windows**

Phone in the **New Project** dialog, as shown in Figure 13. You can leave the default project name unchanged.

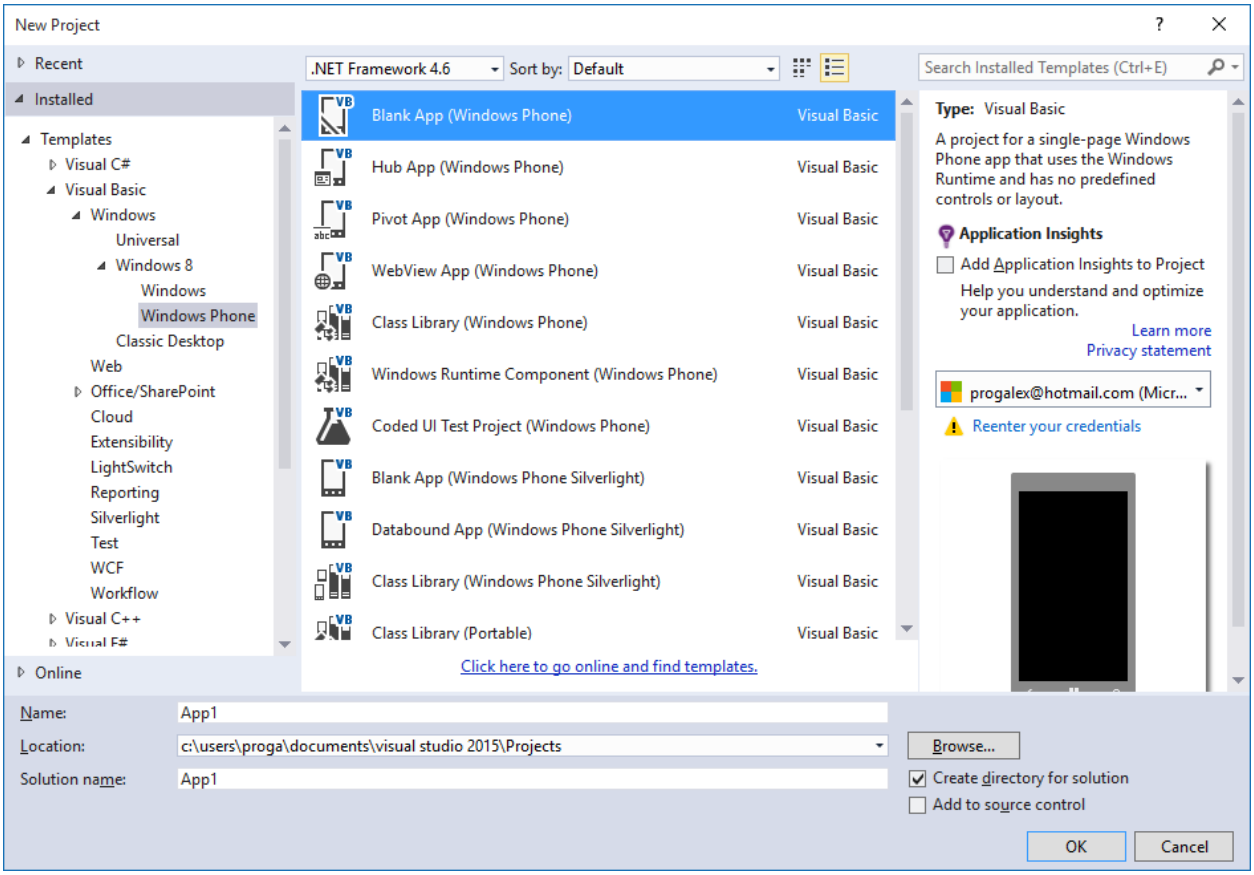

Figure 13: Creating a Windows Phone 8.1 project

When the project is ready, save it, right-click the project name in the **Solution Explorer**, and select **Manage NuGet Packages** from the context menu. At this point, the NuGet Package Manager window appears. In the search box, type **datetime** and search for the **DateTimeAnalyzerVB** package (or DateTimeAnalyzerCS for C#), as shown in Figure 14.

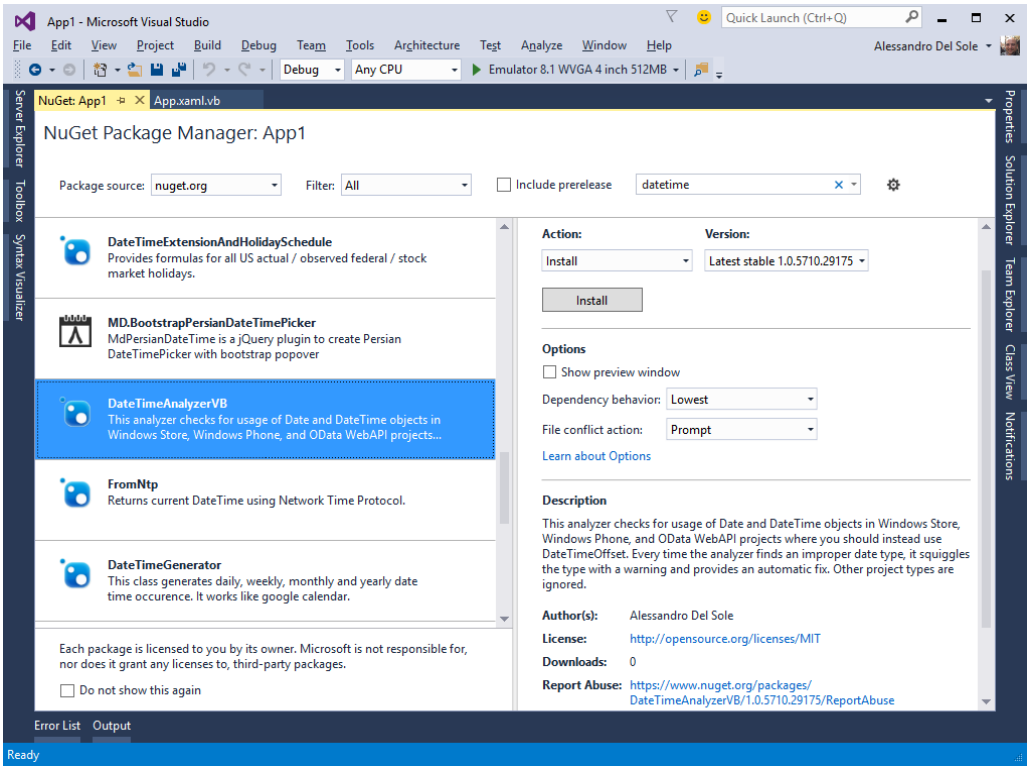

Figure 14: Searching for an analyzer on NuGet

Analyzers for other libraries are also offered as NuGet packages. When you browse for a NuGet package, you can see the description, license agreement (if any), author name, list of available versions, and other information. If you click **Install**, Visual Studio will first display a dialog box containing summary information and the list of dependencies for the selected package, if any. Then you will be asked to accept the license agreement, as shown in Figure 15.

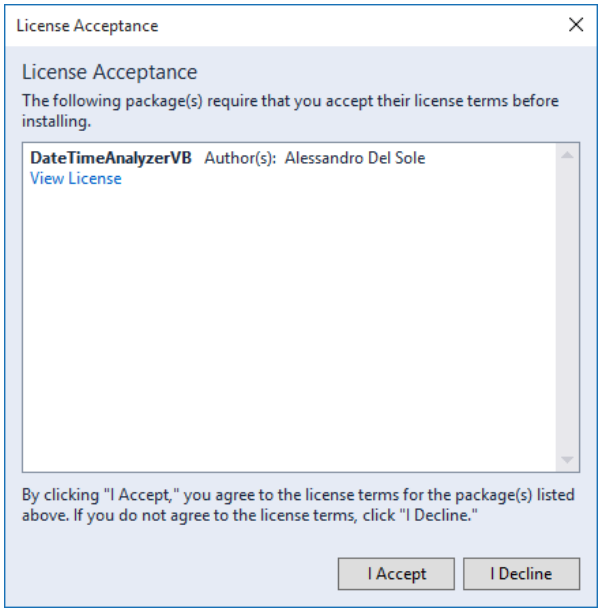

Figure 15: Accepting the package's license agreement

At this point, Visual Studio's integrated NuGet package manager will download the package and place the library in the proper place. The operation's progress is shown in the Output window, and when it completes, you will also see a green check mark symbol near the package name in the NuGet Package Manager window (see Figure 16).

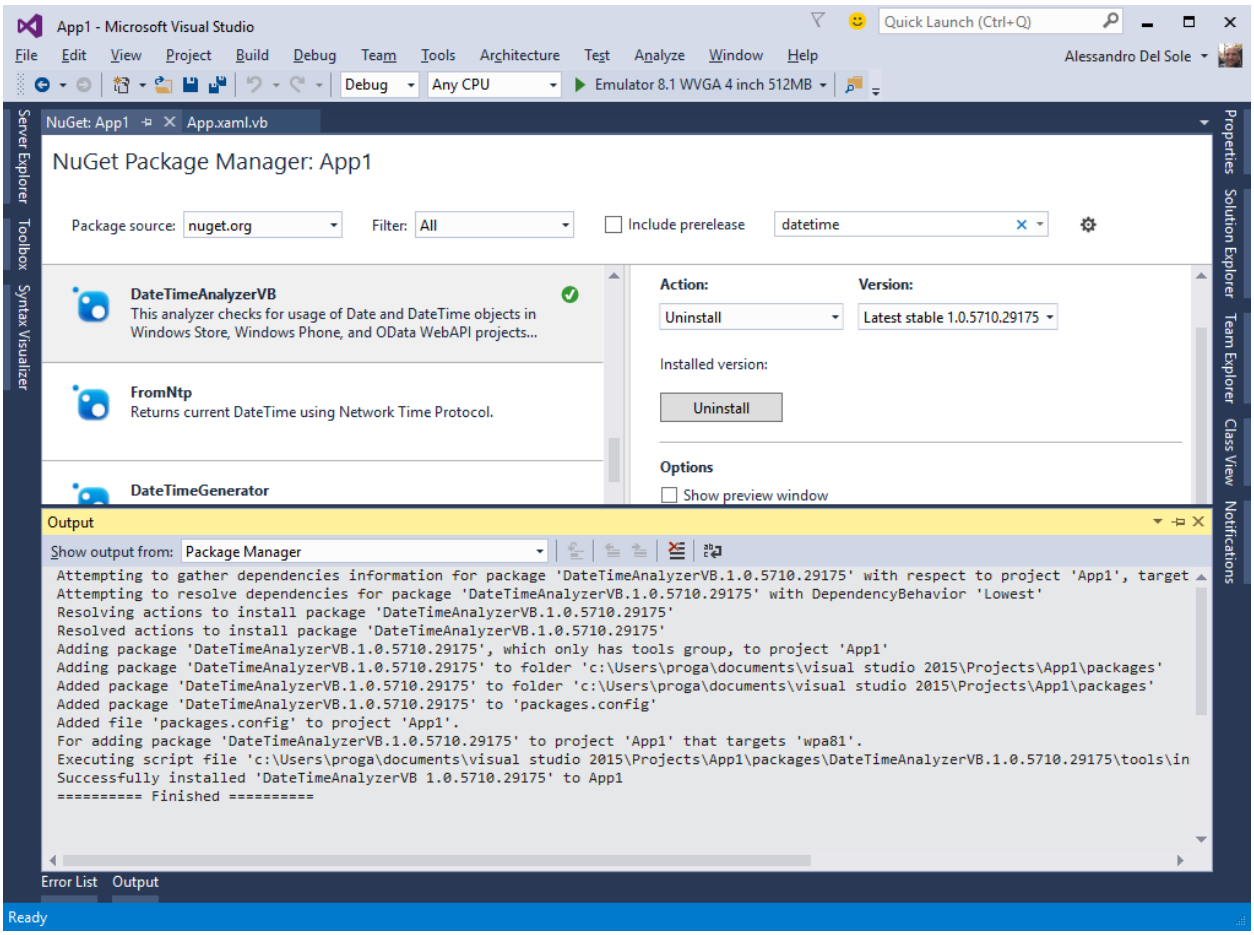

Figure 16: The DateTimeAnalyzerVB analyzer after being downloaded and installed

Tip: This book explains how to install analyzers from NuGet because it is the most common scenario. However, if you want to search for analyzers in the Visual Studio gallery, you must select** Tools **>** Extensions and Updates**, and search for the package online from the** Extensions and Updates **dialog box. This is also exactly what you do when you search for extensions for Visual Studio.

Alternatively, you can install the package from the Package Manager Console. The console can be enabled by selecting **Tools** > **NuGet Package Manager** > **Package Manager Console**. When enabled, type the following in the command line:

```
Install-package DateTimeAnalyzerVB
```

Once the package has been installed, expand the **References** node in the **Solution Explorer**, and then expand the **Analyzers** subnode. The latter contains the list of additional installed analyzers, and here you should see the newly installed **DateTimeAnalyzer** (see Figure 17).

Every analyzer appears as an expandable node. When you expand an analyzer node, you will see the list of diagnostics and refactorings that it exposes.

In this case there is only one diagnostic, called **DTA001: Usage of System.DateTime**. This name is formed from the diagnostic's unique ID and its title. If you click a diagnostic, you will see useful information about it in the Properties window.

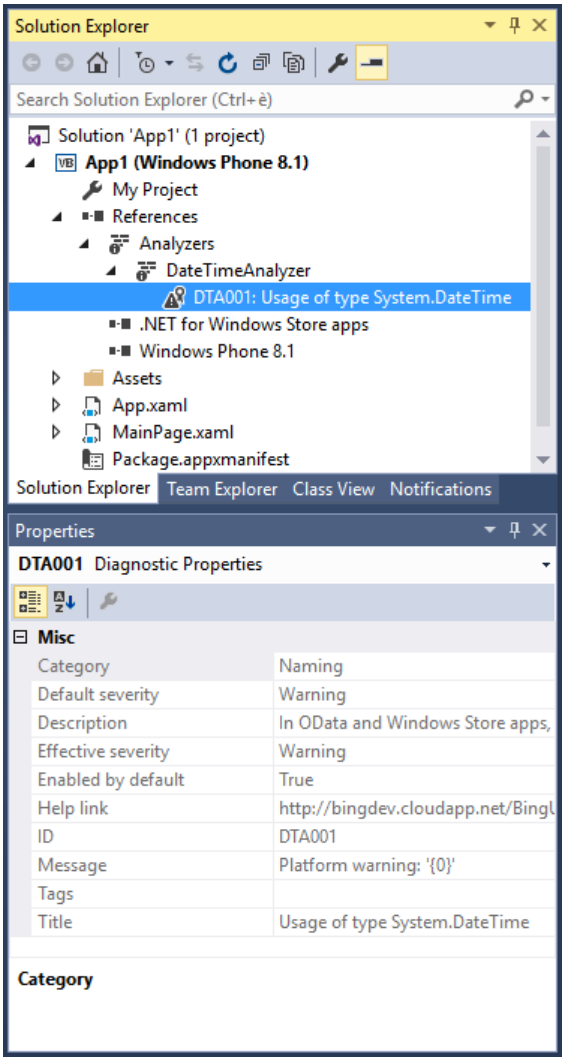

Figure 17: Investigating diagnostics and their properties

Here you can see the diagnostic's description, its severity level (Warning in this case), if it is enabled by default, and other information. Now you will use the downloaded analyzer in practice, and then you will see how to adjust its configuration. Open any code-behind file in the project (that is, a .vb or .cs file according to the language you chose). Then, declare a variable of type **`System.DateTime`** (in Visual Basic you can also use the **`Date`** keyword). As you can see in Figure 18, the code editor detects that you are using a **`DateTime`** object in a project where you may prefer **`DateTimeOffset`**, so it reports a warning and suggests the proper fix.

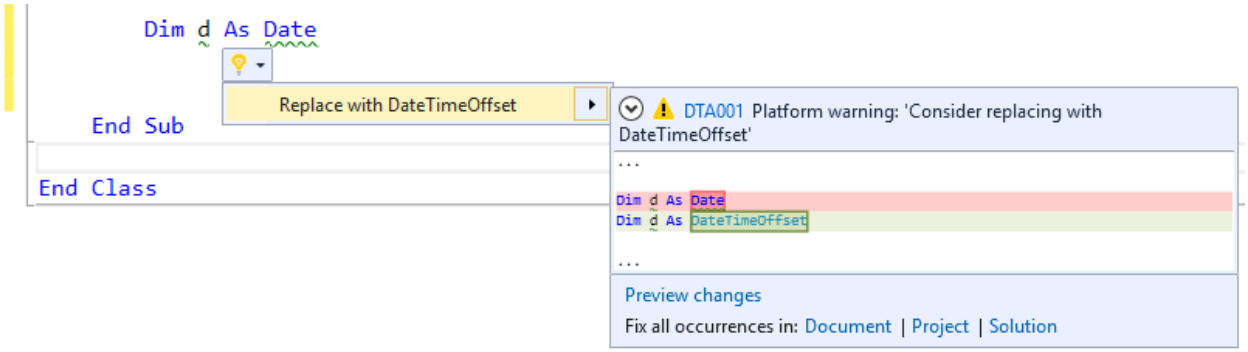

Figure 18: The analyzer detects a code issue and suggests the proper fix

As you can see, installing and using an analyzer is easy because everything integrates seamlessly with the code editor. You can also fine-tune the behavior of both built-in and external analyzers. To accomplish this, double-click **My Project** (for VB) or **Properties** (for C#) in the **Solution Explorer**. When the project's properties window appears, select the **Code Analysis** tab, as shown in Figure 19.

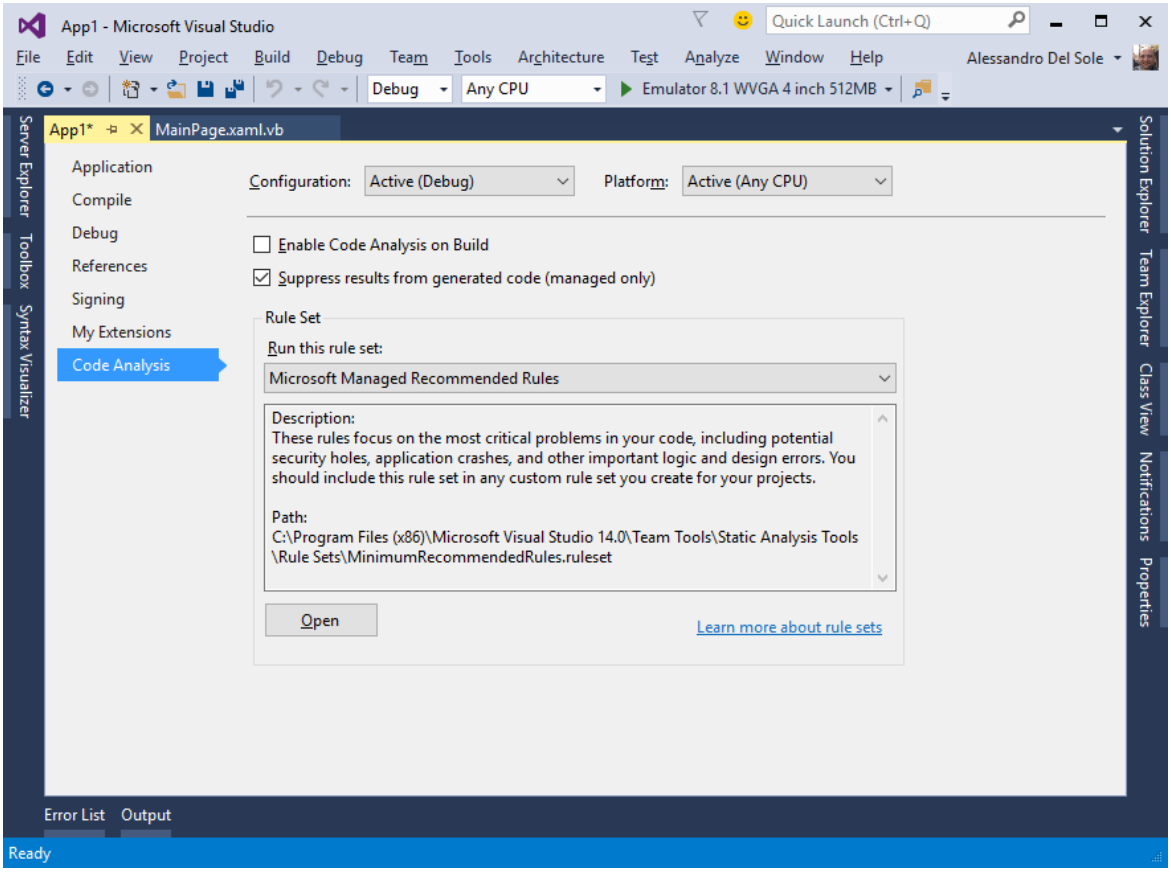

Figure 19: The Code Analysis tab in the project properties window

Do not change the default rule set. Just click **Open**. This will open a window containing the list of analyzers for the current rule set, which includes a number of built-in analyzers. Figure 20 illustrates this.

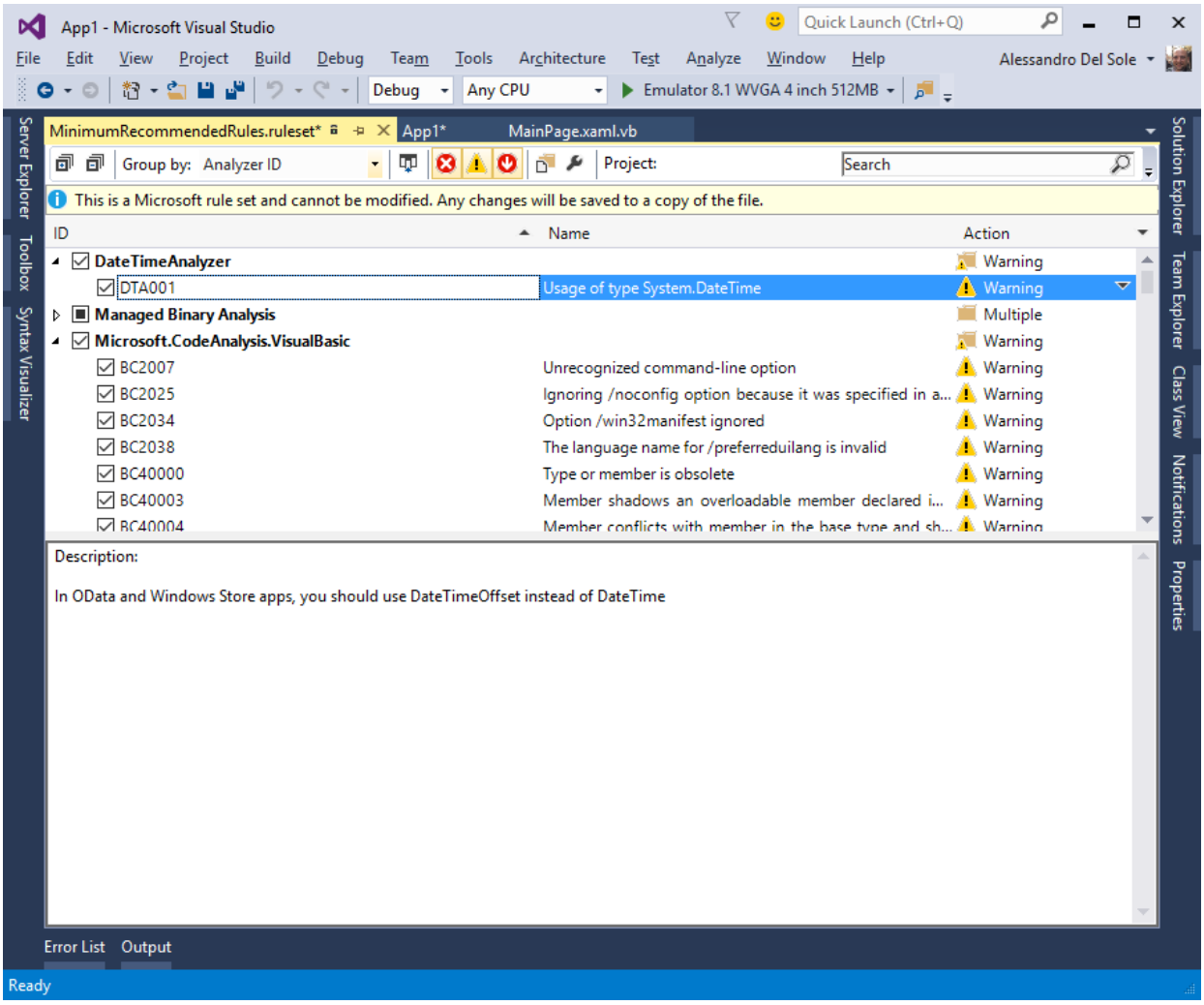

Figure 20: Configuring an analyzer's behavior

Expand the **DateTimeAnalyzer** node. You can enable or turn off the diagnostic by selecting or clearing the check box next to the ID, and you can also change the default severity. For instance, you might disable the diagnostic because you need to work with **`DateTime`** objects, or you can change the default severity from Warning to Error, causing the code editor to display a red squiggle beneath the line of code shown in Figure 18, which would prevent the project from compiling. These adjustments can be also performed against the built-in analyzer shown in the window, but of course you are encouraged to leave the default settings unchanged. Now that you have a better idea of what Roslyn can do and what you can do with analyzers, it's time to move to more complex concepts.

Chapter Summary

This chapter described how the coding experience in Visual Studio 2015 has been enhanced, explaining how many of the features you use in the code editor have been actually rewritten on the .NET Compiler Platform. This empowers the live, static code analysis engine, which detects

code issues as you type and suggests proper fixes. Live code analysis relies on code analyzers and refactorings, both of which are exposed by libraries that leverage the Roslyn APIs.

The very good news is that you are not limited to using built-in analysis rules; you can also download and install third-party analyzers from NuGet or the Visual Studio Gallery. Once installed, analyzers integrate seamlessly with the code editor, and work perfectly detecting code issues based on the custom, domain-specific rules. Additionally, you can fine-tune an analyzer's behavior by enabling and disabling diagnostics, or by changing its severity level.

Chapter 3 Walking through Roslyn: Architecture, APIs, Syntax

The .NET Compiler Platform is built upon several API layers and services. Some of these layers allow exposing the compilers' APIs to consumers and let you take advantage of code analysis to build amazing developer tools. This chapter gives you an important conceptual overview of the .NET Compiler Platform architecture, describing the API layers and services and providing the basis for concepts and topics you will face in the chapters that follow. You will also get started with the fundamental concept of syntax, discovering how the .NET Compiler Platform allows working against source code via managed objects with the help of a visual tool called the Syntax Visualizer, which will become your best friend when coding for Roslyn. That said, take your time to read this chapter carefully. It is all about theory, but it's important to understand before you move on to practice.

The Compiler Pipeline

The .NET Compiler Platform offers open source Visual Basic (VB) and C# compilers with rich code analysis APIs. Code analysis does not mean just searching for code issues and reporting errors in the source code; instead, code analysis means opening compilers to share all the information they have about our code via APIs that developers can use to execute code-related tasks in their applications or tools. To accomplish this, compilers implement an API layer that mirrors the classic compiler pipeline. As shown in Figure 21, the traditional compiler pipeline is made of the following different phases:

1. The parse phase, where the compiler tokenizes the source code and parses it into syntax based on the given language grammar.
2. The symbols and metadata phase, which generates named symbols from declarations and imported metadata.
3. The binder phase, where identifiers are assigned to symbols.
4. The IL emitter phase, where the compiler builds up all the information from the previous phases, emitting an assembly.

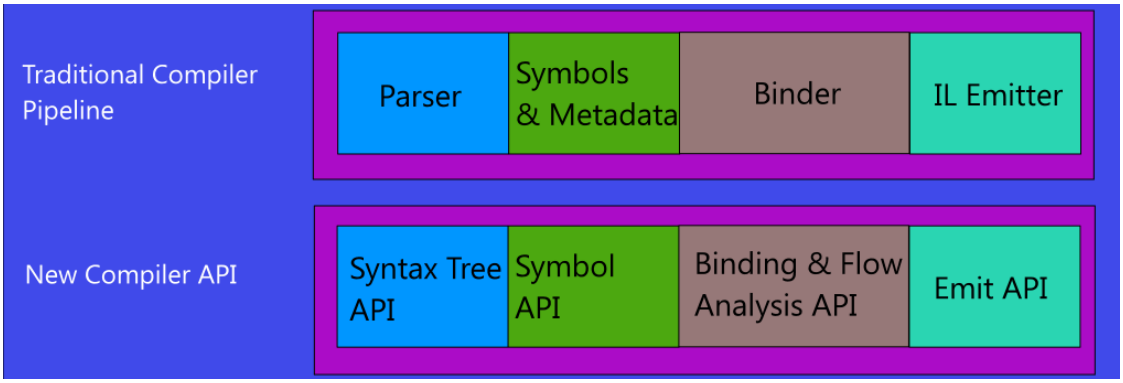

Figure 21: Traditional compiler pipeline and new compiler APIs

In the .NET Compiler Platform, every phase in the compilation process is treated as a separate component surfaced by an object model that exposes information for that phase in the form of .NET objects. The parsing phase is now represented by a **syntax tree** (see "Working with Syntax" later in this chapter). The symbols and metadata phase is represented by a hierarchical symbol table. The binder phase is now represented by an object model that exposes the result of the semantic analysis the compiler performs over the information collected at that point. The IL emitter phase is represented by the Emit API, which produces Intermediate Language (IL) byte codes.

These re-architected phases are also shown in Figure 21, and are referred to as **Compiler APIs**, which will be discussed in more detail shortly. Understanding this change is very important. In fact, it is the first conceptual step you must make to consider the VB and C# compilers as platforms, because it's here that information starts to be open.

The .NET Compiler Platform's Architecture

The .NET Compiler Platform's architecture is made of two main layers and one secondary layer: The Compiler APIs and Workspaces APIs, and the Feature APIs, as shown in Figure 22.

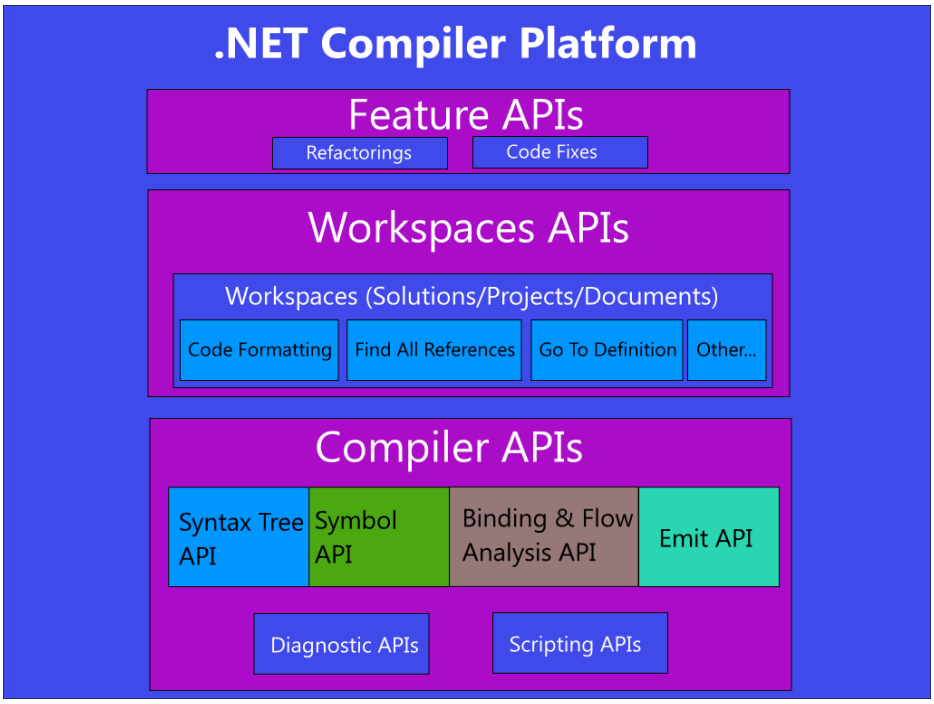

Figure 22: The .NET Compiler Platform's layered architecture

Both layers will be detailed shortly; however, a summary explanation can be useful to give you a starting point. The **Compiler APIs** layer offers an object model that holds information about syntax and semantics exposed at each phase of the compiler pipeline. The compiler not only performs its natural tasks (that is, producing assemblies from source code), but it also shares information about each phase of the pipeline via proper .NET objects. The Compiler APIs also include an immutable snapshot of a single invocation of a compiler, which includes source code files, options, and assembly references.

The second main layer is the **Workspaces APIs** layer, which exposes an object model that holds and organizes all the information in a solution. By using this layer, you can perform code analysis across an entire solution and access the compiler's object model without parsing source code files or configuring compiler options and dependencies.

The **Feature APIs** layer is an additional layer that is specifically designed to offer code fixes and refactorings, and relies on the other layers. Now it is time for a more thorough discussion about the fundamental building blocks of the .NET Compiler Platform and how to work with it.

Assemblies and Namespaces

The .NET Compiler Platform exposes the Compiler APIs and the Workspaces APIs through a set of .NET assemblies. The following is a list of the most important assemblies:

- **`Microsoft.CodeAnalysis.dll`**, which exposes the language-agnostic set of the Compiler APIs.
- **`Microsoft.CodeAnalysis.VisualBasic.dll`**, which exposes the Compiler APIs tailored for Visual Basic.
- **`Microsoft.CodeAnalysis.CSharp.dll`**, which exposes the Compiler APIs tailored for C#.
- **`Microsoft.CodeAnalysis.Workspaces.dll`**, which exposes a common set of the Workspaces APIs.
- **`Microsoft.CodeAnalysis.VisualBasic.Workspaces.dll`**, which exposes Workspaces APIs tailored for Visual Basic.
- **`Microsoft.CodeAnalysis.CSharp.Workspaces.dll`**, which exposes Workspaces APIs tailored for C#.
- **`Microsoft.CodeAnalysis.Common.dll`**, which exposes common utilities for code analysis.
- **`System.Collections.Immutable.dll`**, which exposes special, immutable collections used to work with syntax trees' immutability.

These assemblies are installed in your projects via NuGet. They are automatically installed in any project you create using the templates offered by the .NET Compiler Platform SDK, but you can also download and install them manually to different kinds of projects using the NuGet Package Manager tooling, for instance, if you want to implement code generation inside a stand-alone application. They have corresponding namespaces, such as **`Microsoft.CodeAnalysis`**, **`Microsoft.CodeAnalysis.VisualBasic`**, **`Microsoft.CodeAnalysis.CSharp`**, and so on, which expose the .NET objects required for code analysis.

Tip: The full list of Roslyn assemblies and the full source code can be easily investigated by browsing the online reference source.

The Concept of Immutability

As you will discover in the next sections, the .NET Compiler Platform offers hundreds of types to represent all the information compilers expose. Most of these types, such as **`Solution`**, **`Document`**, **`SyntaxTree`**, and **`SyntaxNode`**, are **immutable**. Immutability is a key concept in Roslyn and provides thread-safety so that multiple developers can work against the same object without any locks or duplication.

For instance, if you want to make changes to a source code file, which is represented by the **`Document`** class, you actually create a new **`Document`** based on the existing one, and supply the required changes. The way you create new objects in code based on existing, immutable objects is shown in practice in Chapters 4 and 5. For now, keep in mind what immutability is, because you will find many references to it throughout this chapter.

The Compiler APIs

The Compiler APIs layer offers an object model related to syntactic and semantic information exposed at each phase of the compiler pipeline. This layer also includes an immutable snapshot of a single invocation of a compiler, which includes assembly references, compiler options, and source code files. There are two distinct compiler APIs for Visual Basic and C#, which are very similar but tailored for high-fidelity to each language. This layer has a fundamental characteristic: it has no dependencies on any Visual Studio components, so any developer can use it in stand-alone applications, too.

Diagnostic APIs

When the compiler analyzes the source code, other than producing an assembly, it may also produce a set of diagnostics that cover syntax and semantics, reporting squiggles, errors, warnings, or informational messages for each code issue. These diagnostics are exposed by the Compiler APIs through the Diagnostic APIs, an extensible set of APIs that allows developers to plug their custom analyzers and diagnostics into the compilation process alongside compiler-defined diagnostics. In this way, this set of APIs can integrate naturally with tools such as MSBuild and Visual Studio 2015, which leverage the Diagnostic APIs to show live squiggles, halt a build on errors, and suggest code fixes.

The Diagnostic APIs have no dependencies on any Visual Studio component, and therefore you could use these libraries to plug an enhanced experience into your developer tools such as code editors. The Diagnostic APIs is the set you will work with more in this book. In fact, both Chapter 4 and Chapter 5 discuss how to create custom, domain-specific analyzers and refactorings that rely on the Diagnostic APIs. Also, any developer who starts working with the .NET Compiler Platform typically begins with the Diagnostic APIs, because they give you a precise idea of what you can do with the Roslyn platform.

Scripting APIs

As part of the Compiler APIs, the .NET Compiler Platform team at Microsoft has created the Scripting APIs, which allow you to compile and execute code snippets and accumulate a runtime execution context. At the time of this writing, this layer is available in the source reference page on GitHub, but has not shipped with .NET Compiler Platform 1.0, and there is no official news about its release.

The Workspaces APIs

The Workspaces APIs layer constitutes the starting point for performing code analysis and refactoring over entire solutions and projects. These APIs organize all the information about the projects in a solution into a single object model and offer direct access to the compiler's object model so that you do not need to parse source code files, configure compile options, or manage projects and their dependencies.

This layer has no dependencies on any Visual Studio components, but it offers a common set of APIs host environments such as IDEs (like Visual Studio 2015) can use to implement code analysis, code refactoring, code formatting and colorization, and more. For instance, Visual Studio 2015 uses these APIs to implement the Find All Reference and Document Outline windows, format and colorize source code in the editor, and perform code generation. The .NET Compiler Platform exposes the Workspaces APIs via specific .NET objects, which are described in the next section and reexamined in Chapter 8 with more examples.

Understanding Workspaces, Solutions, Projects, and Documents

A workspace is an instance of the **`Microsoft.CodeAnalysis.Workspace`** object, and is the representation of an MSBuild solution as a collection of projects, each with a collection of documents where each document represents a .vb or .cs source code file. More specifically:

- A solution is an immutable model of projects and documents that is represented by an instance of the **`Microsoft.CodeAnalysis.Solution`** type. Because of immutability, the solution model can be shared without locking or duplication. You can get an instance of the current solution in the workspace with the **`Workspace.CurrentSolution`** property, which is of type **`Solution`**, and which never changes because of its immutable nature. You can modify a solution by creating a new one based on an existing solution plus the required changes. The newly created solution must be applied back to the workspace in order to reflect the supplied changes.
- A project is an immutable instance of the **`Microsoft.CodeAnalysis.Project`** class and is a collection of all the source code documents, assembly references, project-to-project references, parse options, and compilation options. From **`Project`**, you can access information about the compilation without the need to parse source code files or iterating project references, and you can also access information such as the solution name, assembly name, version, and output file path.
- A document represents a single source code file and is represented by an instance of the **`Microsoft.CodeAnalysis.Document`** class. This allows accessing the source text of the file and elements such as the syntax tree and the semantic model, both of which are described shortly.

Because the Workspaces APIs have no dependencies on Visual Studio 2015, every host environment, such as custom IDEs, can access the object model of a solution. This allows host environments to retrieve all the solution information, or perform code-related tasks such as code analysis, refactorings, or formatting over an entire solution, a project, or a single document.

Tip: You can also create stand-alone workspaces that are disconnected from the host environment or use workspaces inside applications that do not act as host environments. You will see an example of the latter scenario in Chapter 8, "Workspaces, Code Generation, Emit".

Working with Syntax

With the Roslyn APIs, you can analyze and generate Visual Basic and C# source code. This can happen against source code opened inside a Visual Studio solution, source code that an application produces at runtime, source code from files on disk, and, more generally, source code from any location.

Source code is actually text. Whether you need to parse source text to generate binary objects or round-trip back to source text from binary objects, you need a structured way to represent the source code. Programming languages like C# and Visual Basic have their own lexicon, syntax, semantics, and everything that is needed to write a program.

In the .NET Compiler Platform, the lexical and syntactic structure of the source code is represented by **syntax trees**. Syntax trees are represented by instances of the `Microsoft.CodeAnalysis.SyntaxTree` class and hold the source information in full fidelity, so they are important for the following reasons:

- They allow rearranging the source code in a managed, .NET way instead of working with pure text.
- They allow Visual Studio 2015 to elaborate the syntactic structure of the source code in a solution.
- Compilation, code analysis, code generation, and code refactoring are built upon them.
- They represent every single item the compiler detects in the source code, such as code blocks, syntactical constructs, comments, and white spaces, exactly as they were typed. As an implication, a syntax tree can be used to round-trip source code back to the text it was parsed from.

As you'll learn shortly, syntax trees are represented by .NET objects. This provides a way to replace a syntax tree with a new one in a managed way, without working against pure text. An important fact to consider about syntax trees at this point is that once you create a syntax tree or you get an instance of an existing tree, this never changes. In fact, syntax trees are immutable. Now you will get started with a visual tool called the Syntax Visualizer, which allows you to get more familiar with syntax elements.

Investigating Syntax with the Syntax Visualizer

I have stated several times that in Visual Studio 2015 the code editor has been rebuilt on top of the .NET Compiler Platform. This means that Visual Studio itself and managed compilers use syntax trees to parse and analyze the source code you write.

For a better and deeper understanding of the syntactic structure of source code, the .NET Compiler Platform SDK offers a nice tool window called the **Syntax Visualizer**, which you enable via **View** > **Other Windows** > **Syntax Visualizer**. The Syntax Visualizer makes it easier to browse the source code in a Roslyn-oriented way, showing the syntax trees and child elements a code block is made of. For instance, consider the code in Code Listing 2, which is referred to as a console application.

Code Listing 2

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SyntaxVisualizerDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime oneDate = new DateTime(2015, 8, 19);

            //Formatting a string with interpolation
            Console.WriteLine($"Today is {oneDate.ToShortDateString()}");
            Console.ReadLine();
        }
    }
}
```

The code is very simple; it just creates a new instance of the **`System.DateTime`** object and prints the date properties to the Console window. This small piece of code is enough to help you understand many concepts about syntax trees.

Enable the Syntax Visualizer and then click anywhere in the code editor. When you click, the Syntax Visualizer starts representing the content of the code file in a hierarchical way, as shown in Figure 23.

The Syntax Visualizer is made of two areas: **Syntax Tree** and **Properties**. The Syntax Tree area provides a hierarchical view of the syntax for the current file. Every item in the hierarchy is colored according to its meaning: Blue represents a syntax node, green represents a syntax token, and maroon represents a syntax trivia (all of these elements are discussed shortly). You can click the Legend button to get an explanation about the colors.

The Properties area shows details about the currently selected item in the syntax tree. As you will discover in Chapter 4, "Writing Code Analyzers," the Properties area is also useful to determine which .NET types map the selected syntax node, token, or trivia depending on its kind, and it makes it easier to understand what .NET objects you need to parse or generate code blocks. You will use the Syntax Visualizer many times in this book, so do not worry if something is not clear at this point.

As you can see in Figure 23, at the first level of the hierarchy, there is the CompilationUnit. A CompilationUnit represents the whole structured content of a source code file (.cs or .vb). If you expand such an element, you can find many nested elements, each representing a part of the source code.

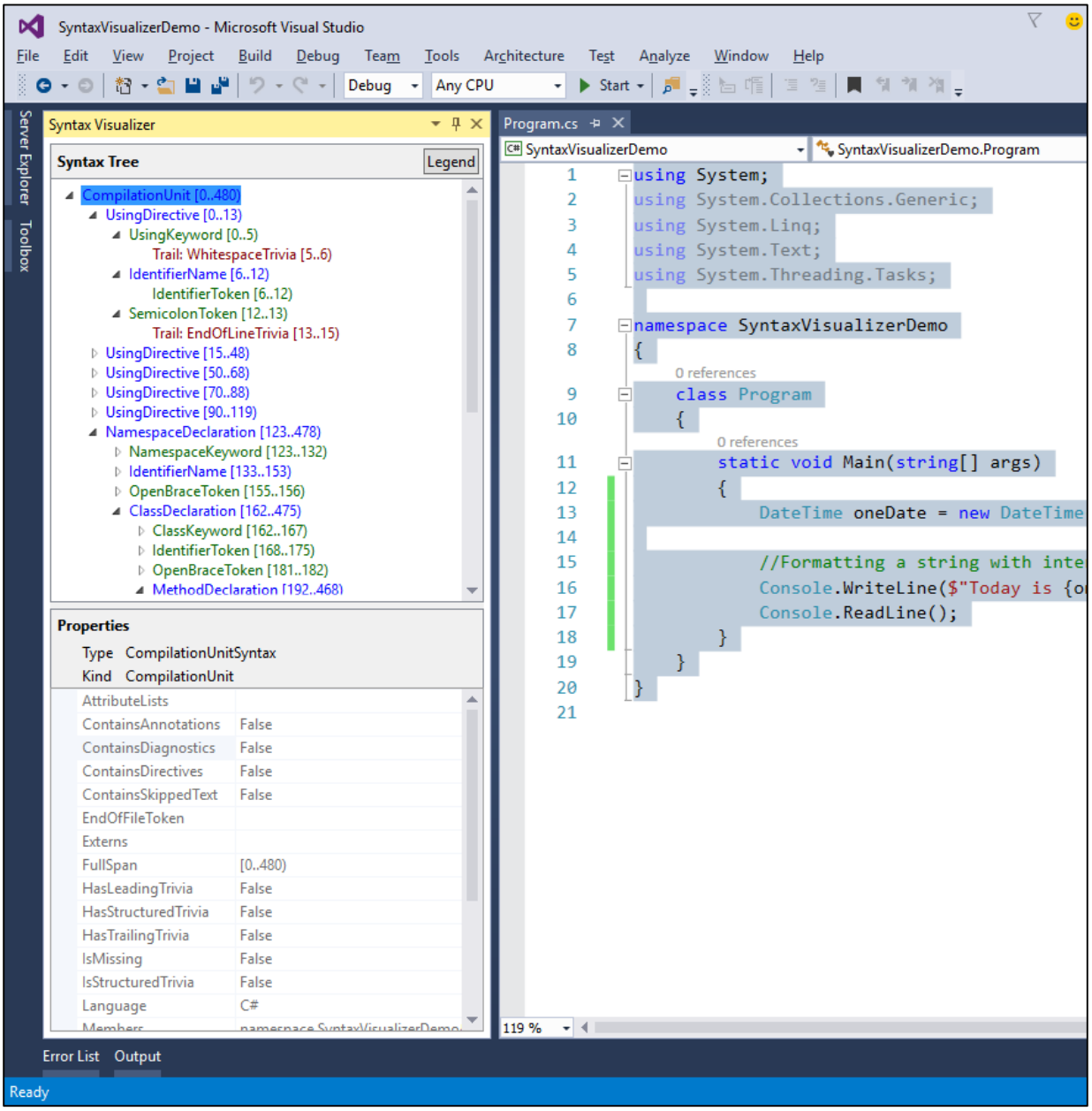

Figure 23: The Syntax Visualizer allows inspecting the syntactic structure of source code

For example, if you expand the CompilationUnit you find, among the others, a NamespaceDeclaration that represents a namespace definition and its content. If you expand the NamespaceDeclaration, you see a ClassDeclaration and its syntax elements. In this case, you see how the **`Program`** class is defined. Within the ClassDeclaration, if you click and expand the MethodDeclaration element, you can see all the parts that constitute the **`Main`** method definition (see Figure 24).

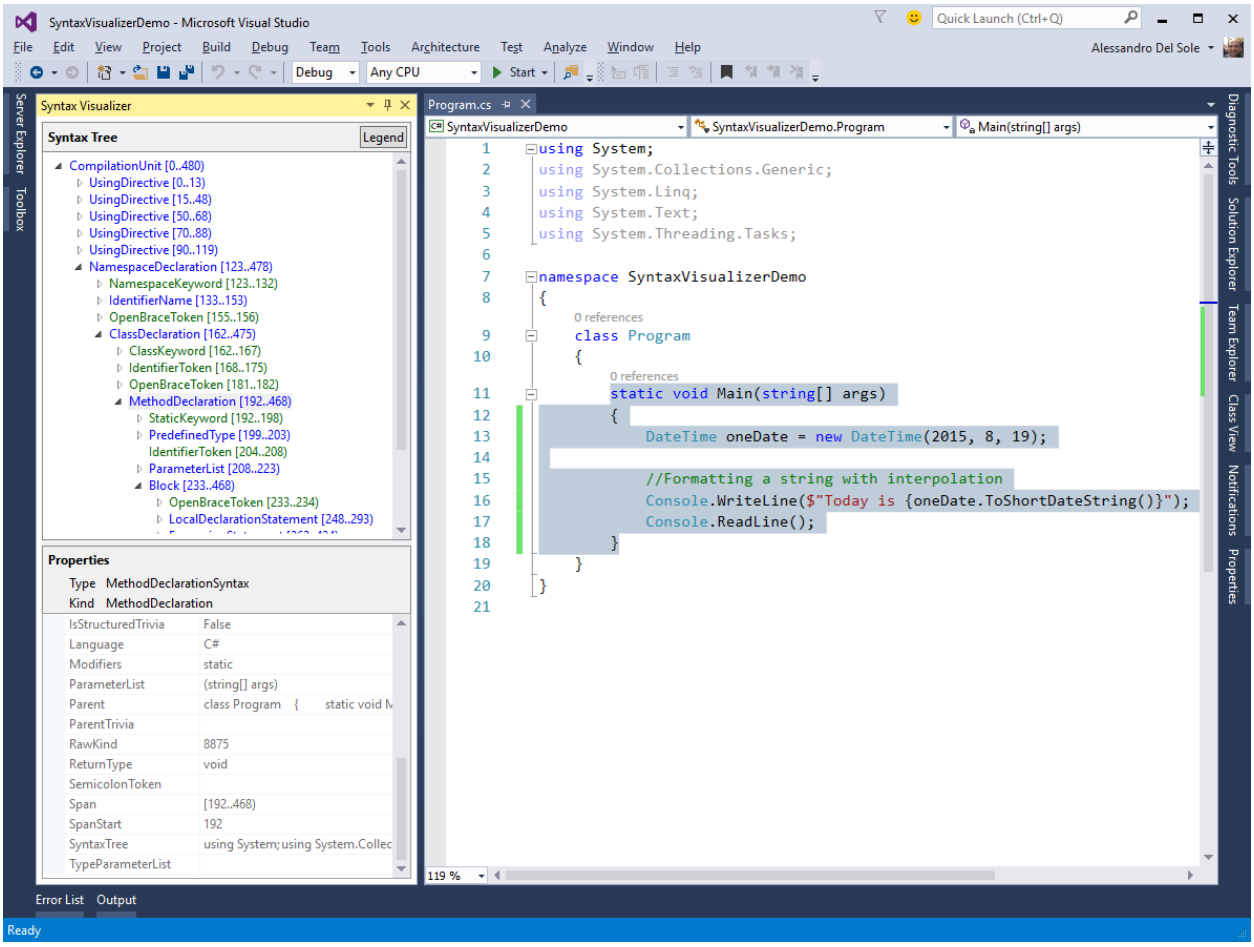

Figure 24: Understanding a method block definition

As you can see, the MethodDeclaration represents a method definition as a whole. For each of these elements, you will find an object that represents every keyword, every white space, every punctuation element, every type name, and so on, exactly as typed in the source text. If you walk through the syntax elements that compose the method definition, you can find one called LocalDeclarationStatement, which represents the entire line of code that declares and instantiates an object of type **`System.DateTime`**. The LocalDeclarationStatement has nested elements, each representing a type name, a white space, the object initialization expression, and the semicolon. This is shown in Figure 25.

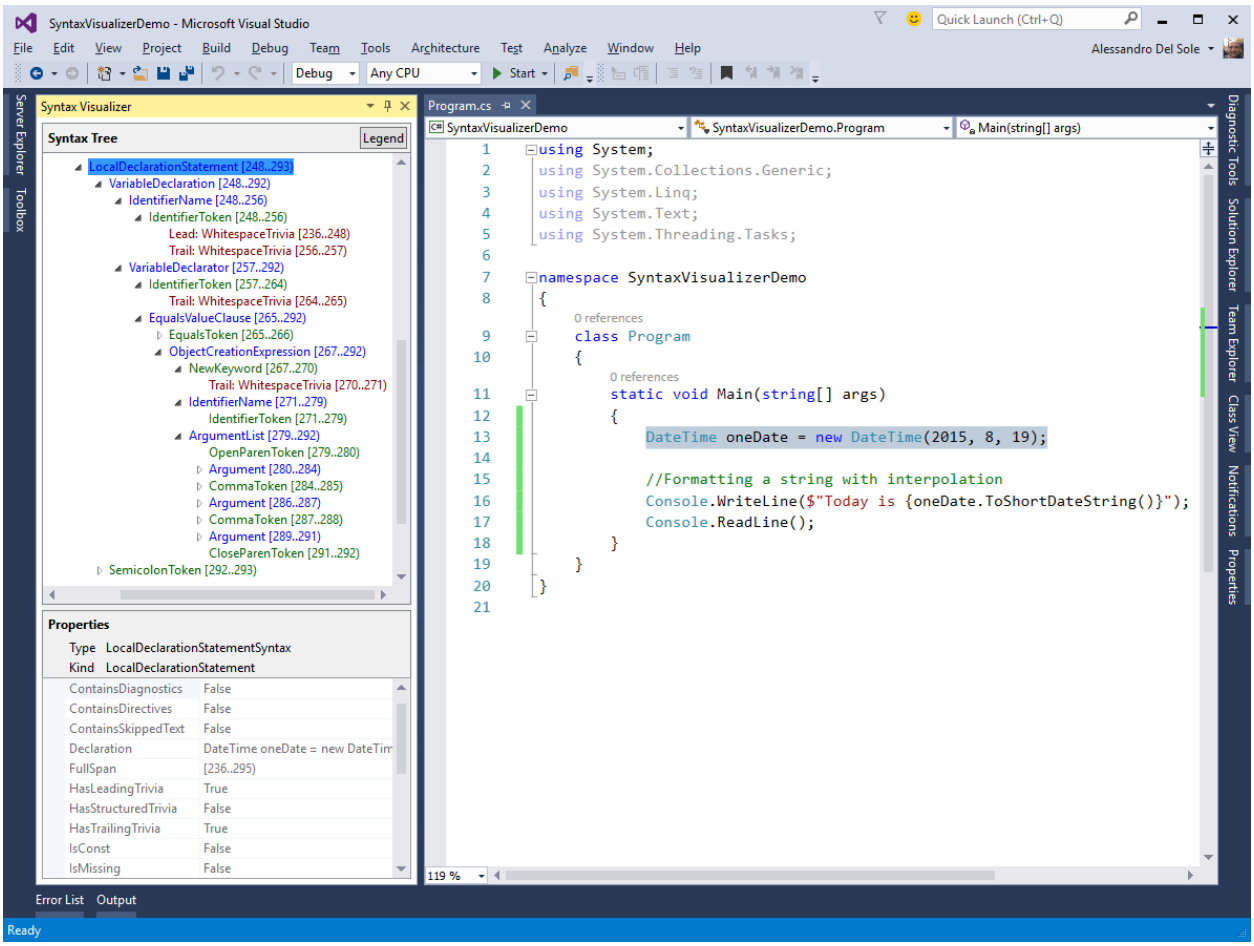

Figure 25: Understanding how an object is declared and instantiated

If you select the line of code that contains the **`Console.WriteLine`** statement, you will see a syntax element called ExpressionStatement, which is made of nested elements such as the method invocation, the interpolated string, the invocation to **`ToShortDateString`**, and all the necessary elements. This is demonstrated in Figure 26.

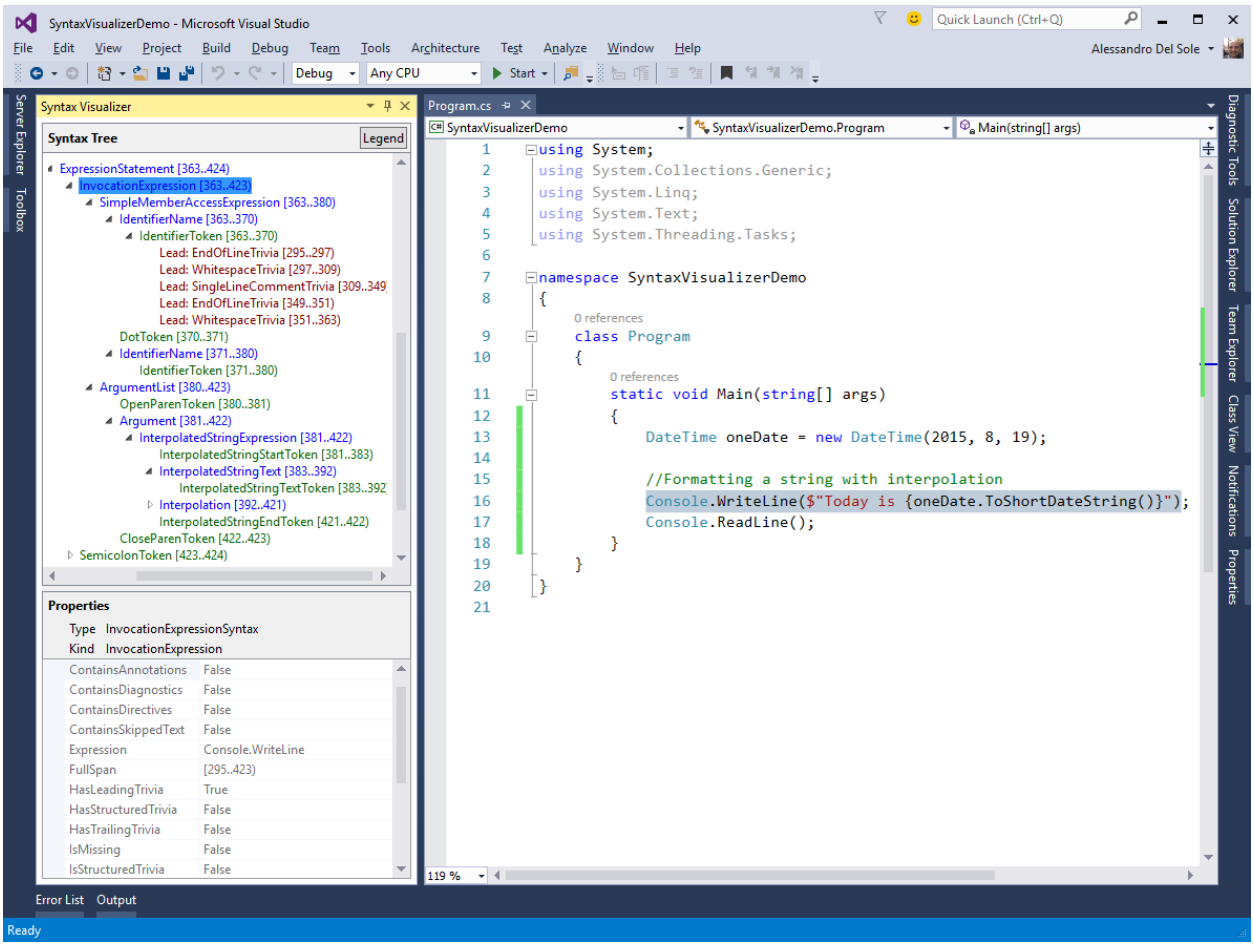

Figure 26: Understanding method invocations with arguments

Not all of the syntax elements involved in the hierarchy have been mentioned, and this is intentional. In fact, I want to encourage you to practice with the Syntax Visualizer on your own by selecting and expanding many of the nodes in the syntax tree to get more familiar with the Roslyn representation of the source text. In the sections that follow, you will get more detailed information about syntactical elements, and in Chapters 4 and 5 you will use many syntax elements in practice with the help of the Syntax Visualizer.

For now, click in the editor, expand the Syntax Tree, and try to check the meaning of syntax elements in the Properties area of the Syntax Visualizer. You will discover how intuitive it is understanding the syntactical representation of the code in the .NET Compiler Platform.

Understanding Syntax Elements

Syntax trees have a hierarchical structure with descendant nodes, called syntax nodes, that have child syntax elements, as you can see by using the Syntax Visualizer tool. This section describes syntax nodes and other syntax elements.

 ***Tip: All the syntax elements described in this section will be revisited and discussed in practice in Chapters 4 and 5.***

Syntax Nodes

Syntax nodes represent syntactical constructs within syntax trees, such as expressions, class blocks, statements, and clauses. The most basic representation of a syntax node is an instance of the **`Microsoft.CodeAnalysis.SyntaxNode`** class, which is the base class for more specialized objects representing different kinds of constructs. For instance, the **`ClassStatementSyntax`** class represents a syntax node related to a class declaration in Visual Basic, and the base class in its inheritance hierarchy is **`SyntaxNode`**.

 ***Tip: You can explore a syntax node's complete inheritance hierarchy with the Object Browser tool window.***

In the Syntax Visualizer tool, syntax nodes are represented in blue. If you look back at Figure 24, you can see some examples of syntax nodes, such as UsingDirective and NamespaceDeclaration.

The following is a list of characteristics of syntax nodes:

- They always have other child nodes and tokens.
- They expose a **`ChildNodes`** method, which returns a sequential list of nested syntax nodes, but does not return syntax tokens.
- They expose an immutable **`Parent`** property, which allows accessing the parent node.
- They expose **`Descendant`** methods like **`DescendantNodes`**, **`DescendantTokens`**, and **`DescendantTrivia`**, which allow accessing child elements of a syntax node and return a list of nodes, a list of tokens, and a list of trivia, respectively, within the subnode.
- Each specialized syntax node class allows accessing named properties. For instance, the **`NamespaceDeclarationSyntax`** class, which represents a namespace's syntax node, has **`NamespaceKeyword`** and **`Name`** properties, representing the **`namespace`** keyword and the namespace name, respectively.

Syntax nodes are primary elements in the syntax hierarchy. You will work with specialized syntax node classes very often when writing code analyzers and refactorings, as you will see in more detail in the next two chapters.

Syntax Tokens

Syntax tokens represent identifiers, punctuation, literals, and keywords. They are never parents of syntax nodes or other syntax tokens. If you take a look back at Figure 24, you can see how the NamespaceKeyword element is a syntax token; in the Syntax Visualizer tool, syntax tokens are represented in green.

Syntax tokens are represented by the **Microsoft.CodeAnalysis.SyntaxToken** structure. It is a value type for efficiency purposes, and unlike syntax nodes, there is only one **SyntaxToken** structure for all kinds of tokens with a number of properties that are common to every value they represent. Among others, **SyntaxToken** exposes the following properties:

- **Value** (of type **Object**), which returns the raw value of the object it represents.
- **ValueText** (of type **String**), which returns the result of invoking the **ToString()** method on **Value**.

The difference between the two properties can be summarized as follows: Imagine you have an integer literal token, such as 1000. **Value** returns the exact decoded integer value **1000**. **ValueText** can recognize Unicode characters identified by an escape sequence; therefore, it can contain **1000**, but also **&H3E8** (hexadecimal) or **&O1750** (octal). These three tokens have different text, but they have the same value (1000).

Syntax Trivia

Syntax trivia represent portions of the source text that the compiler ignores to emit the IL, such as white spaces, comments, XML comments, preprocessing directives, and end-of-line terminators. They exist to maintain full fidelity with the original source text, but they are not included in a syntax tree's child nodes. So, when you generate a new tree based on an existing syntax tree, you must explicitly include trivia. A trivia is represented by the **Microsoft.CodeAnalysis.SyntaxTrivia** structure, another value type. You will use the **LeadingTrivia** and **TrailingTrivia** collections to investigate a node's trivia.

Kinds

More often than not, you need to identify the exact syntax node type you are working with, or disambiguate syntax node types that share the same node class. To make things easier, syntax nodes, syntax tokens, and syntax trivia expose a **Kind** extension method that returns an integer value from the **SyntaxKind** enumeration, which identifies the exact syntax element represented. **SlashToken** and **ForKeyword** are examples of syntax elements represented by values in the enumeration. They also expose an **IsKind** extension method that returns a Boolean value indicating if the syntax node is of the specified syntax kind.

Tip: There are separate *SyntaxKind* ***enumerations for VB and C#. In VB, you use the*** *Microsoft.CodeAnalysis.VisualBasic.SyntaxKind* ***enumeration, whereas in C# you use the*** *Microsoft.CodeAnalysis.CSharp.SyntaxKind* ***enumeration.***

Spans

Especially when producing diagnostics, the host environment must know where to place error and warning squiggles or highlights in the code editor. Syntax nodes, tokens, and trivia know their position within the source code and the number of characters they consist of. This information is referred to as **spans**, and is represented via the **TextSpan** object. Syntax nodes expose the **Span** and **FullSpan** properties, both of type **TextSpan**. Both represent the text span from the start of the first token in the syntax node's subtree to the end of the last token, but **Span** does not include the span of any trivia, whereas **FullSpan** does.

Errors

Roslyn can expose a full syntax tree that can be round-tripped back to the source text even if it contains syntax errors. In this particular case, the parser uses one of the following techniques to create a syntax tree:

- If the parser expects a particular kind of token but does not find it, it may add a missing token into the syntax tree in the location where the token was expected. This **`SyntaxToken`** has an empty span, and its **`IsMissing`** property returns **`true`**.
- The parser skips tokens until it finds one where it can continue to parse. Skipped tokens are collected as a token's trivia whose kind is **`SkippedTokens`**.

You can easily demonstrate this by adding any intentional error in your code and seeing how the Syntax Visualizer highlights in red the lines of code that contain diagnostics, maintaining a hierarchical syntax tree anyway.

Introducing Semantics

While syntax trees represent the lexical and syntactic structure of the source code, they are not enough to represent everything that is necessary to produce a program. In fact, it is necessary to apply the language rules to produce meaningful results, and there is the need to determine what is being referenced. For instance, you might have types and members with the same name across the source code, and therefore you need a way to determine what type or member an identifier refers to. Additionally, a program can have references to compiled libraries, and even if the source code is not available, a program can still use types they define. Semantics encapsulate all the information required to produce a program into an object model made of different components, each described in the next sections.

Understanding Compilation

The compilation represents everything needed to compile a C# or Visual Basic program and is represented by an object of type **`Microsoft.CodeAnalysis.Compilation`**. Compilation includes source files, but also assembly references and compile options. It represents each declared type, member, or variable as a **symbol**. The **`Compilation`** class exposes a number of methods that make it easier to find and relate symbols, no matter if they have been declared in the source code or imported from an assembly. A compilation is also immutable, which means that it cannot be changed once created. You can, however, create a new compilation based on an existing one, making the necessary changes, such as adding assembly references.

Understanding Symbols

Types, namespaces, methods, parameters, properties, fields, events, variables, and every distinct element either declared in the source code or imported from an assembly is represented by a **symbol**. Symbols contain information that the compiler determines from the source or from the assembly's metadata, including referenced symbols. Each kind of symbol is represented by specialized interfaces derived from **`ISymbol`**, each tailored for the symbol it needs to represent.

For instance, the **`IMethodSymbol`** interface represents information that the compiler collected about a method. Its **`ReturnType`** property determines the method's return type. Symbols are particularly important in the .NET Compiler Platform, and the Syntax Visualizer allows you to get a visual representation of symbols. To demonstrate this, right-click a syntax node, such as the **ExpressionStement** node associated with the **`Console.ReadLine()`** statement, and then select **View Symbol (if any)**. At this point, the **Properties** area of the Syntax Visualizer shows full symbol information for the selected syntax node, as shown in Figure 27.

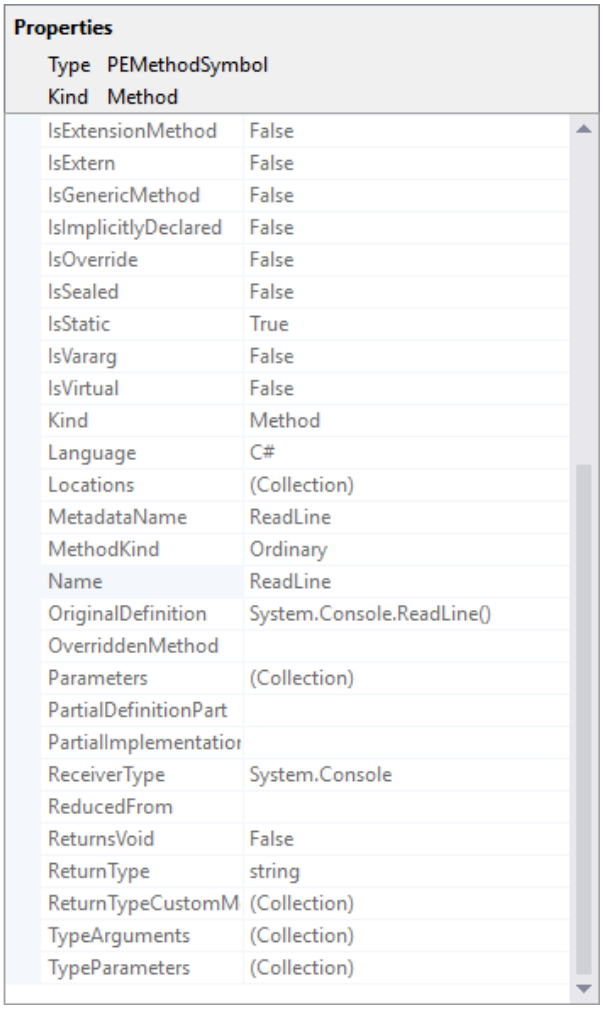

Figure 27: Viewing symbol information in the Syntax Visualizer

The **`Compilation`** class exposes a number of methods and properties that help you find symbols from source or imported metadata. Symbols are conceptually similar to the Reflection APIs in the CLR type system, but they are the representation of language concepts, not CLR concepts. It is worth mentioning that symbols are not available for syntax tokens and syntax trivia. Symbols can save you a lot of time, and you will understand their importance as you get more familiar with the Roslyn APIs.

Introducing the Semantic Model

All the semantic information for a source file is represented by a **semantic model**, mapped by an object of type **`Microsoft.CodeAnalysis.SemanticModel`**. This model helps you discover the following information:

- All the diagnostics (errors and warnings).
- Symbols referenced at a specific location.
- Result types for any expressions.
- The flow of variables across regions in the source code.

Each syntax tree in a compilation owns an instance of the **`SemanticModel`** class. The semantic model is also very useful, and you will use it many times in both analyzers and refactorings.

Chapter Summary

This chapter offered a high-level overview of the .NET Compiler Platform's architecture and APIs. You first saw how the classic compiler pipeline has been rewritten through a mirrored series of .NET objects. Then you saw how the architecture is made of two main layers: The Compiler APIs and the Workspaces APIs. You learned about the concept of immutability, which is so important in the .NET Compiler Platform, and you learned about workspaces, which give you an option to access an object model that represents a solution with its projects and documents.

You then moved to the core of the .NET Compiler Platform, which is syntax, and learned how to use the Syntax Visualizer tool and what syntax elements are available to represent the source code in a structured, hierarchical, and managed way. Finally, you were introduced to semantics, which represent everything needed to compile a program, and provide an object model you can use to access the compilation properties and every declared symbol.

Theory is enough for now. You have all the necessary foundations required to start using the Roslyn APIs in code, and you will begin writing your first code analyzer in the next chapter.

Chapter 4 Writing Code Analyzers

Visual Studio has always offered integrated analysis to detect code issues. In previous versions, some code issues were detected by the background compilers as you typed, and the IDE immediately reported errors and warnings with squiggles in the code editor, showing the related error or warning messages in the Error List window. In specific scenarios, the code editor also offered an option to edit the code issues with proper fixes, but the majority of errors and other code issues could only be detected when compiling a solution. Also, code analysis was limited to rules coded at Microsoft, but it was impossible to enhance the code analysis engine with domain-specific rules, at least in the code editor.

The .NET Compiler Platform brings revolutionary changes. With the new Compiler APIs, not only does live static code analysis detect any code issues as you type, but it can be extended with custom, domain-specific analysis rules. In this chapter, you'll learn how to build a code analyzer for custom syntax rules that plugs into the Visual Studio 2015 code editor, as well as many other concepts about syntax.

 ***Note: When coding analyzers, you often use asynchronous methods and the async/await pattern. This requires you to have at least basic knowledge of how an asynchronous method works, the `async` and `await` keywords, and the concept of cancellation tokens. The MSDN Library has a specific page about this programming pattern, which you are strongly encouraged to read before going on with this chapter if you are new to this way of coding.***

Writing Code Analyzers

A code analyzer is a library that includes one or more custom rules for detecting domain-specific errors and code issues as you type. One rule and all the related information the compiler collects about it is referred to as a **diagnostic**. A code analyzer (or simply analyzer) can also suggest proper fixes within the light bulb, as discussed in Chapter 2. In this chapter, you will create an analyzer that detects if the source code improperly assigns the **`System.DateTime`** type to a variable in Windows Store and Windows Phone projects, where you should use the **`System.DateTimeOffset`** type instead, at least when you work with user controls that allow manipulating dates.

 ***Note: Examples in this book do not provide the ultimate solution to a specific problem or completely address every possible situation. In most cases, they will need to be improved to fit into real-world scenarios. However, they have been architected in a way that makes it easier for you to get started with syntax, to learn the right approach, and to understand all the concepts related to building developer tools with the Roslyn APIs. After reading this chapter and the next one, you will have all the required fundamentals to explore additional APIs and to create your custom analysis rules for any scenario.***

Creating an Analyzer with Code Fix Project

To create an analyzer, select **File** > **New Project**, and then in the **Extensibility** node under the language of your choice, select the **Analyzer with Code Fix (NuGet + VSIX)** project template, as shown in Figure 28.

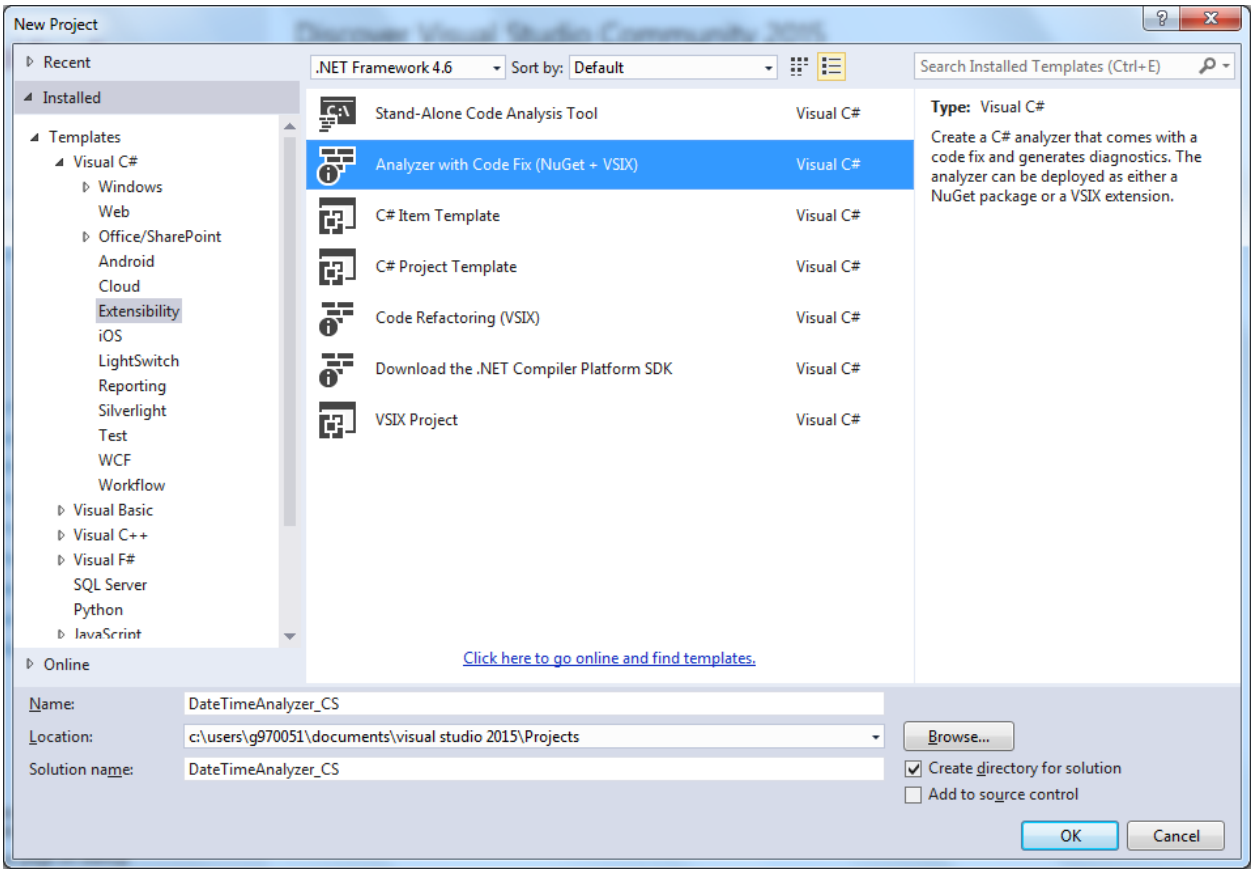

Figure 28: Creating an analyzer project

Type **DateTimeAnalyzer_CS** as the C# project name, or **DateTimeAnalyzer_VB** as the Visual Basic project name, and then click **OK**. In the following explanations, I'm using the C# project name, but the same procedure applies to Visual Basic (VB).

The selected project template generates a sample analyzer whose instructional purpose is detecting and fixing lowercase letters within type names. I will not discuss the auto-generated sample; instead, I will show you how to start from scratch, but you can definitely take a look at the source code as an exercise. With this template, Visual Studio 2015 generates a solution with three projects:

- **DateTimeAnalyzer_CS (Portable)**, a portable class library that implements the analyzer. This project contains two files: DiagnosticAnalyzer.cs (or .vb), which is responsible for implementing the analysis logic and raises the proper errors or warnings, and CodeFixProvider.cs (or .vb), which is where you implement code fixes that integrate into the light bulb. These are conventional file names that you can rename with more meaningful names. This is something you definitely want to do when adding multiple analyzers and code fixes to a single project.

- **DateTimeAnalyzer_CS.Test**, a project that contains unit tests to test your analyzer in a separate environment. This particular project is not discussed in this book because I can't assume you have knowledge of unit testing and of test projects. It is not mandatory to create and deploy analyzers, though.
- **DateTimeAnalyzer_CS.Vsix**, which packages the analyzer into a VSIX package. The generated VSIX package can be used to publish your analyzer to the Visual Studio Gallery (see Chapter 7), but most importantly at this point, it is used by Visual Studio to install the analyzer into the experimental instance of the IDE, which is necessary to debug your analyzers (and code refactorings).

Understanding Syntax Elements to Work With

Before you implement the analysis logic, you must first understand which syntax elements you need to work with. To do this, you use the Syntax Visualizer tool discussed in Chapter 3. For instance, consider Figure 29, where you can see the code declares a variable called **`oneDate`**, of type **`System.DateTime`**.

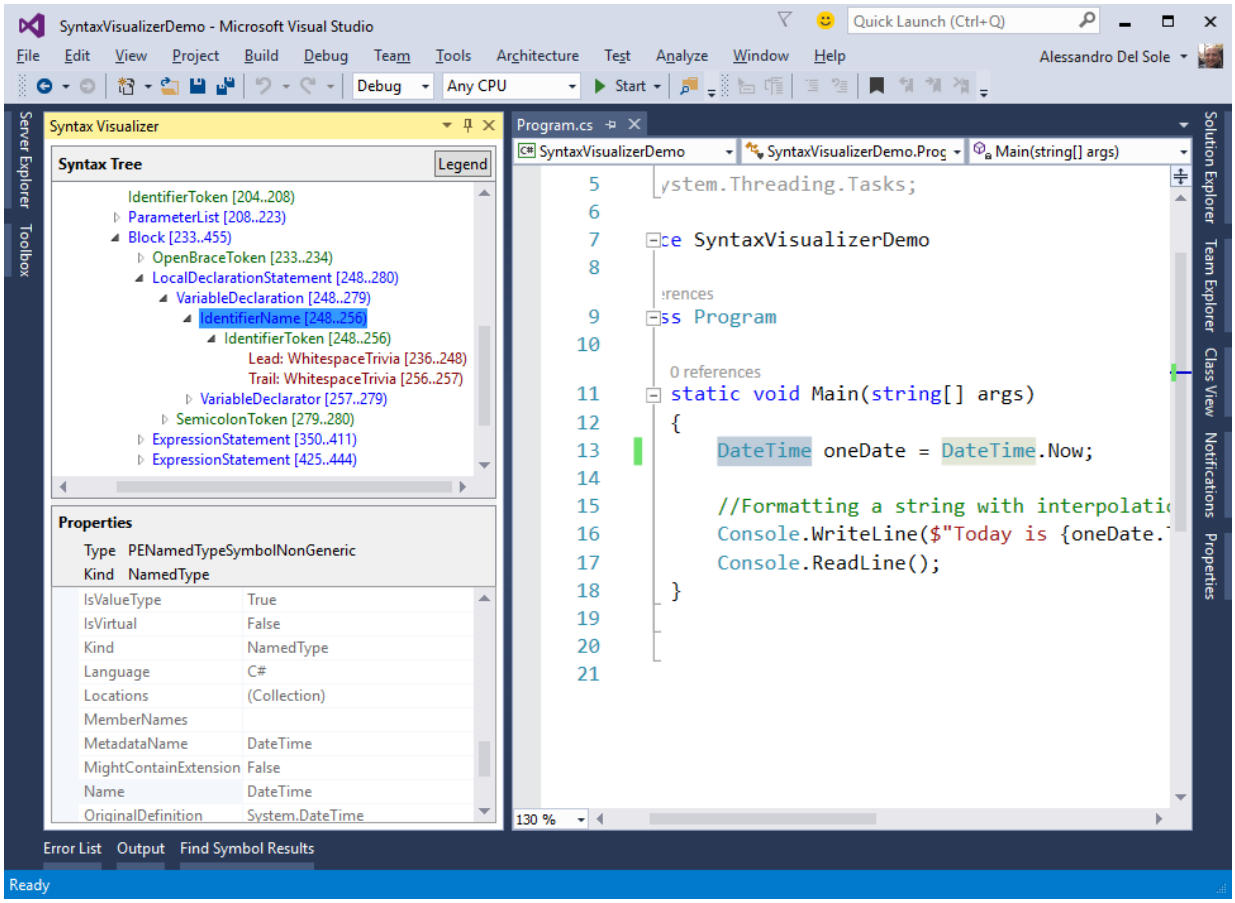

Figure 29: Using the Syntax Visualizer to investigate a syntax node

Suppose your analyzer needs to report a warning every time it detects the **`DateTime`** type in a Windows Store app. The first thing you need to do is understand how **`DateTime`** is represented with Roslyn. As you can see from Figure 29, it is represented by an IdentifierName syntax node

of type **`IdentifierNameSyntax`**. The Properties area is useful to understand properties for this kind of syntax node, but it might be more useful to discover if this identifier has a matching symbol, which provides much more information. To accomplish this, right-click **IdentifierName** and then select **View Symbol (if any)**. Now the Properties area shows symbol information, such as the syntax kind, the containing assembly, the base type, and the type name, which you will use shortly. This kind of symbol is called PENamedTypeSymbolNonGeneric, and is represented by a .NET interface called **`INamedTypeSymbol`**.

The analysis logic in the analyzer will:

- Focus only on IdentifierName syntax nodes. This improves the analysis performance by avoiding the need of investigating an entire syntax tree, excluding all the other kinds of syntax nodes from the analysis.
- Get the symbol information for these kinds of syntax nodes.
- Focus only on named type symbols.
- Detect if the analyzed named type symbol is called DateTime, and if so, the analyzer will create a new diagnostic and report a warning so that the code editor will display a green squiggle beneath the named type.

Tip: As a general rule, you can determine the exact syntax node you want to analyze with the Syntax Visualizer by selecting a sample line of code so that you can exclude all the other syntax nodes from the analysis. This is the very first step in architecting efficient analysis logics.

Understanding Common Elements

Domain-specific rules, diagnostics, and the analysis logic are defined in a class that inherits from **`Microsoft.CodeAnalysis.Diagnostics.DiagnosticAnalyzer`** and must be decorated with the **`DiagnosticAnalyzer`** attribute, whose argument is a constant from the **`LanguageNames`** class, representing the target programming language (**`CSharp`** or **`VisualBasic`**).

Visual Studio automatically generates a class that meets these requirements when you create an analyzer project whose name is based on the project name. Open the DiagnosticAnalyzer.cs (or .vb) file, and consider Code Listing 3, which already includes some edits to the auto-generated code.

Code Listing 3 (C#)

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class DateTimeAnalyzer_CSAnalyzer : DiagnosticAnalyzer
{
    public const string DiagnosticId = "DTA001";

    // You can change these strings in the Resources.resx file.
    // If you do not want your analyzer to be localize-able,
    // you can use regular strings for Title and MessageFormat.
    private static readonly LocalizableString Title =
```

```
        new LocalizableResourceString(nameof(Resources.AnalyzerTitle),
            Resources.ResourceManager, typeof(Resources));
    private static readonly LocalizableString MessageFormat =
        new LocalizableResourceString(nameof(Resources.
        AnalyzerMessageFormat),
        Resources.ResourceManager, typeof(Resources));
    private static readonly LocalizableString Description =
        new LocalizableResourceString(nameof(Resources.
        AnalyzerDescription),
        Resources.ResourceManager, typeof(Resources));
    private const string Category = "Naming";
```

Code Listing 3 (Visual Basic)

```
<DiagnosticAnalyzer(LanguageNames.VisualBasic)>
Public Class DateTimeAnalyzer_VBAnalyzer
    Inherits DiagnosticAnalyzer

    Public Const DiagnosticId = "DTA001"

    ' You can change these strings in the Resources.resx file.
    'If you do not want your analyzer To be localize-able,
    'you can use regular strings For Title And MessageFormat.
    Private Shared ReadOnly Title As LocalizableString =
        New LocalizableResourceString(NameOf(My.Resources.AnalyzerTitle),
        My.Resources.ResourceManager,
        GetType(My.Resources.Resources))
    Private Shared ReadOnly MessageFormat As LocalizableString =
        New LocalizableResourceString(NameOf(My.Resources.
        AnalyzerMessageFormat),
        My.Resources.ResourceManager,
        GetType(My.Resources.Resources))
    Private Shared ReadOnly Description As LocalizableString =
        New LocalizableResourceString(NameOf(My.Resources.
        AnalyzerDescription), My.Resources.ResourceManager,
        GetType(My.Resources.Resources))
    Private Const Category = "Naming"
```

 ***Note: Visual Studio names the analyzer's root namespace and the auto-generated class based on the project name, adding the Analyzer suffix. So, like in the current example, the namespace is DateTimeAnalyzer_CS, but the class name is DateTimeAnalyzer_CSAnalyzer. If you think this double suffix might be confusing or you simply don't like it, don't add the Analyzer suffix when you name a new project or rename the analyzer class to remove this suffix.***

The **`DiagnosticId`** constant represents a unique identifier for the analyzer, and in Code Listing 3, the default ID has been changed with a custom one. Of course, you are free to write a

different ID. Together with the value of **`MessageFormat`**, the diagnostic ID is displayed in the Error List window. **`DiagnosticId`**, **`MessageFormat`**, and **`Title`** are also displayed in the diagnostic's tooltip that appears when the pointer hovers over a squiggle or when a light bulb over a code issue is enabled. **`Description`** provides a way to describe the diagnostic in greater detail.

By default, **`Title`**, **`MessageFormat`**, and **`Description`** are of type **`LocalizableResourceString`**, and their value is supplied via the project's default resources through the Resources.resx file. This approach makes it easier to implement the analyzer's localization, but you can definitely use regular strings if you do not want an analyzer to be localized. To understand how this is defined, in the **Solution Explorer**, double-click the **Resources.resx** file (for Visual Basic, you must first enable the **Show All Files** view and then expand **My Project**) so that the resource editor appears, as shown in Figure 30.

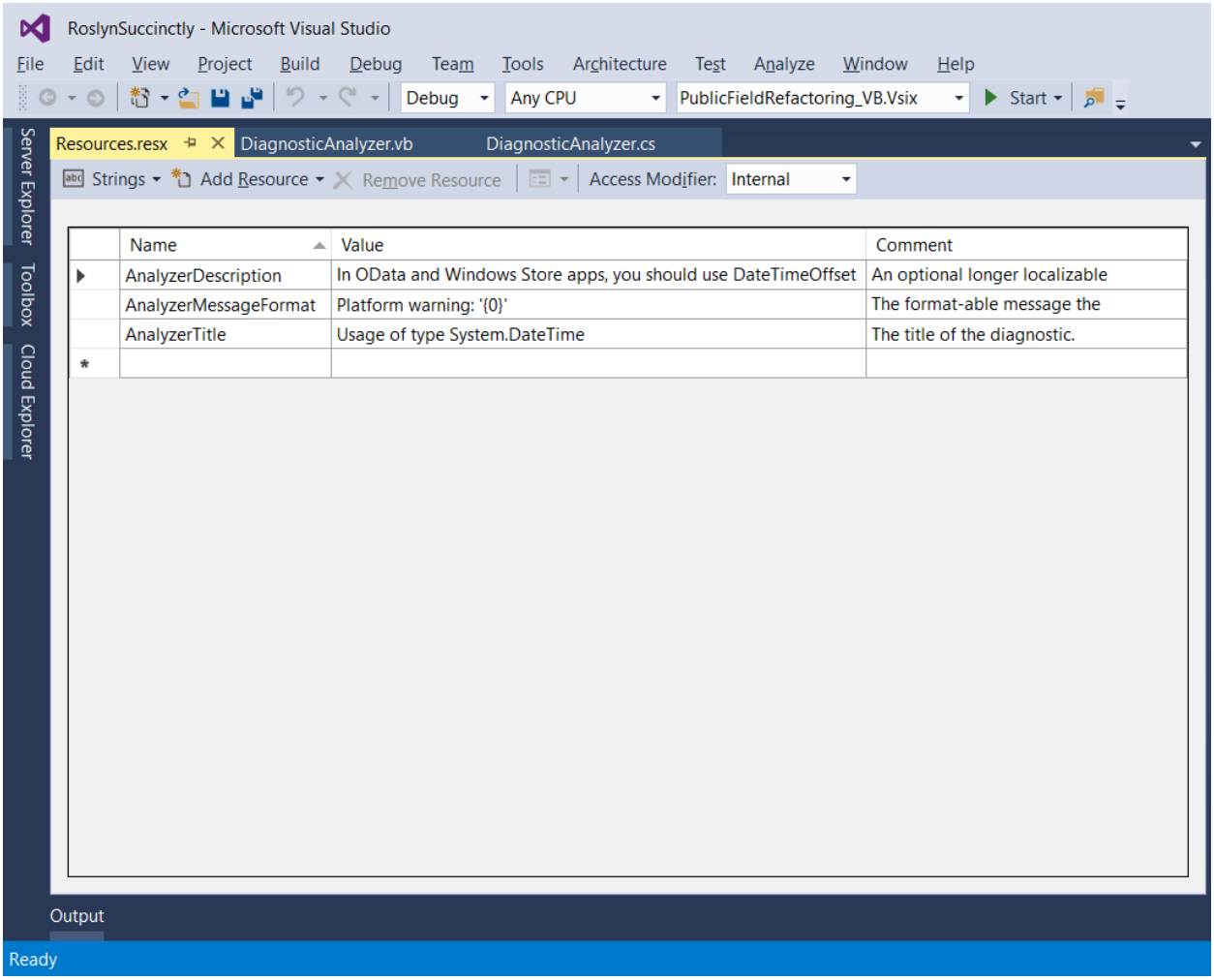

Figure 30: Editing resource strings

There are three string resources defined by default. Replace the **Value** column with the content shown in Figure 30 ("In OData and Windows Store Apps, you should use DateTimeOffset"), which shows proper strings for the current example. The **`Category`** constant represents the category the analyzer belongs to, such as naming, syntax, maintainability, design, and so on.

The category you choose does not influence the way an analyzer works. It is instead a way to give developers additional information.

Tip: A useful list of category names can be found at https://msdn.microsoft.com/en-us/library/ee1hzekz.aspx.

For each rule you add to the analyzer, you need a so-called diagnostic descriptor, which is an instance of the **`Microsoft.CodeAnalysis.DiagnosticDescriptor`** class. This describes a diagnostic in a way that can be understood by tools such as Visual Studio. A **`DiagnosticDescriptor`** instance exposes properties containing the following information about a diagnostic:

- The values of **`DiagnosticId`**, **`MessageFormat`**, **`Title`**, **`Category`**, and **`Description`**, represented by properties with matching names.
- If it must be enabled by default, or if it must be enabled manually in the rule set editor. This piece of information is held by the **`IsEnabledByDefault`** Boolean property.
- The diagnostic's severity level, which is represented by the **`DefaultSeverity`** property, and whose value can be a value from the **`Microsoft.CodeAnalysis.DiagnosticSeverity`** enumeration, such as **`Warning`**, **`Error`**, **`Info`**, or **`Hidden`**. Remember that only the **`Error`** severity level prevents a project from being compiled until a code issue is fixed.
- An optional URI to the analyzer's online documentation (if any), which developers will open by clicking the diagnostic ID in the Error List. This is represented by the **`HelpLinkUri`** property. Note that if no custom URI is supplied, Visual Studio 2015 will start a Bing search based on the diagnostic ID and title. At this time, there is no option to change the default search engine (unless you provide a custom URI).

A diagnostic descriptor is exposed via an immutable property called **`SupportedDiagnostics`**, which returns a **`System.Collections.Immutable.ImmutableArray<DiagnosticDescriptor>`**. Normally, this immutable array contains only the diagnostic descriptor's instance. The implementation for both the diagnostic descriptor and the immutable array is shown in Code Listing 4.

Code Listing 4 (C#)

```
private static DiagnosticDescriptor Rule =
        new DiagnosticDescriptor(DiagnosticId,
        Title, MessageFormat, Category,
        DiagnosticSeverity.Warning,
        isEnabledByDefault: true,
        description: Description,
        helpLinkUri:
    "https://github.com/AlessandroDelSole/RoslynSuccinctly/wiki/DTA001");

public override ImmutableArray<DiagnosticDescriptor>
     SupportedDiagnostics { get { return ImmutableArray.Create(Rule); } }
```

Code Listing 4 (VB)

```
Private Shared Rule As New DiagnosticDescriptor(DiagnosticId,
        Title, MessageFormat, Category,
        DiagnosticSeverity.Warning,
        isEnabledByDefault:=True,
        description:=Description,
        helpLinkUri:=
     "https://github.com/AlessandroDelSole/RoslynSuccinctly/wiki/DTA001")

Public Overrides ReadOnly Property SupportedDiagnostics As _
       ImmutableArray(Of DiagnosticDescriptor)
   Get
       Return ImmutableArray.Create(Rule)
   End Get
End Property
```

Now it is time to implement the actual analysis logic.

Implementing the Analysis Logic

The analysis logic requires two fundamental steps: writing a method that will perform the code analysis over a given syntax node, referred to as **action**, and registering the action at the analyzer's startup so that the analyzer can respond to compiler events, such as finding syntax nodes or declaring symbols.

You can implement and register multiple actions for a syntax node. This can be useful if you want to detect multiple issues and offer multiple fixes. You register actions inside the **`Initialize`** method, which is the main entry point of an analyzer. This method takes an argument called **`context`**, of type **`AnalysisContext`**, which represents the context for initializing an analyzer. The **`AnalysisContext`** type exposes a number of methods that you use to register actions, as described in Table 1.

Table 1: Methods for registering actions

Method Name	Description
`RegisterSyntaxTreeAction`	Registers an action that is executed after a code file is parsed.
`RegisterSyntaxNodeAction`	Registers an action that is executed at the completion of the semantic analysis of a syntax node.
`RegisterSymbolAction`	Registers an action that is executed at completion of the semantic analysis over a symbol.

Method Name	Description
RegisterSemanticModelAction	Registers an action that is executed after a code document is parsed.
RegisterCompilationStartAction	Registers an action that is executed when compilation starts.
RegisterCompilationEndAction	Registers an action that is executed when compilation completes.

As I mentioned previously (see Figure 29), you must analyze an IdentifierName node, which is represented by the **IdentifierNameSyntax** type, and therefore a syntax node. For this reason, you need to register an action invoking the **RegisterSyntaxNodeAction** method, as shown in Code Listing 5.

Code Listing 5 (C#)

```
public override void Initialize(AnalysisContext context)
{
    context.RegisterSyntaxNodeAction(AnalyzeDateTime,
            SyntaxKind.IdentifierName);
}
```

Code Listing 5 (VB)

```
Public Overrides Sub Initialize(context As AnalysisContext)
    context.RegisterSyntaxNodeAction(AddressOf AnalyzeDateTime,
                                     SyntaxKind.IdentifierName)
End Sub
```

The first parameter of **RegisterSyntaxNodeAction** is a delegate that will perform the actual code analysis, whereas the second argument is the syntax kind (see Chapter 3 for details about kinds). This is specified by picking one of the values from the **SyntaxKind** enumeration, in this case **IdentifierName**, which represents an IdentifierName syntax node.

Note: The `SyntaxKind` enumeration exposes almost 800 values, so it's impossible to provide a description for each value in this book. However, IntelliSense always does a great job describing values, and you can just right-click the enumeration in the code editor and select Go To Definition to explore all the available syntax kinds.

It's finally time to analyze a syntax node. This is done in a method we call **AnalyzeDateTime**, whose code is shown in Code Listing 6. Read the comments carefully. Additional explanations will follow shortly.

Code Listing 6 (C#)

```
        private void AnalyzeDateTime(SyntaxNodeAnalysisContext context)
        {

            // Get the syntax node to analyze
            var root = context.Node;

            // If it's not an IdentifierName syntax node,
            // return
            if (!(root is IdentifierNameSyntax))
            {
                return;
            }

            // Convert to IdentifierNameSyntax
            root = (IdentifierNameSyntax)context.Node;

            // Get the symbol info for
            // the DateTime type declaration
            var dateSymbol = context.SemanticModel.
                GetSymbolInfo(root).Symbol as INamedTypeSymbol;

            // If no symbol info, return
            if (dateSymbol == null)
            {
                return;
            }

            // If the name of the symbol is not
            // DateTime, return
            if (!(dateSymbol.MetadataName == "DateTime"))
            {
                return;
            }

            // Create a diagnostic at the node location
            // with the specified message and rule info
            var diagn = Diagnostic.Create(Rule,
                root.GetLocation(),
                "Consider replacing with DateTimeOffset");

            // Report the diagnostic
            context.ReportDiagnostic(diagn);
        }
```

Code Listing 6 (VB)

```
Private Sub AnalyzeDateTime(context As SyntaxNodeAnalysisContext)

    'Get the syntax node to analyze
    Dim root = context.Node

    'If it's not an IdentifierName syntax node,
    'return
    If TypeOf (root) Is IdentifierNameSyntax
       'Convert to IdentifierNameSyntax
        root = CType(context.Node, IdentifierNameSyntax)
    Else
        Return
    End If

    'Get the symbol info for
    'the DateTime type declaration
    Dim dateSymbol =
        TryCast(context.SemanticModel.
                GetSymbolInfo(root).Symbol,
                INamedTypeSymbol)

    'If no symbol info, return
    If dateSymbol Is Nothing Then
        Return
    End If

    'If the name of the symbol is not
    'DateTime, return
    If Not dateSymbol.MetadataName = "DateTime" Then
        Return
    End If

    'Create a diagnostic at the node location
    'with the specified message and rule info
    Dim diagn =
        Diagnostic.Create(Rule, root.GetLocation,
                    "Consider replacing with DateTimeOffset")

    'Report the diagnostic
    context.ReportDiagnostic(diagn)
End Sub
```

Retrieving a Syntax Node's Information

The **AnalyzeDateTime** method has a parameter of type **SyntaxNodeAnalysisContext**, which represents the context for a syntax node and holds information about a syntax node from the compiler's point of view. This object exposes a property called **Node**, of type **SyntaxNode**, which

represents the actual syntax node you work with. As you learned in Chapter 3, **SyntaxNode** is at the root of the syntax hierarchy, and therefore, you need a more specialized object. So the code attempts to convert the syntax node into an **IdentifierNameSyntax** object, which is the one representing the **DateTime** type. If the conversion fails, it means that the current node is not an IdentifierName, so the execution of the method terminates. If the conversion succeeds, then the current node is an IdentifierName, and the code attempts to retrieve symbol information for the node.

Retrieving Symbol Information

Retrieving symbol information for the node is accomplished by invoking the **GetSymbolInfo** method over the **SemanticModel** of the syntax node's context, and then converting the **Symbol** property into an object of type **INamedTypeSymbol**, which derives from **ISymbol** and is designed to represent named types like **DateTime**.

If the result of this invocation and conversion is null, it means that there is no symbol information associated with this IdentifierName node, and the method execution terminates. If it's not null, the code checks if the value of the **MetadataName** property is **DateTime**. **MetadataName** contains the type name for the symbol being examined. In this example, if **MetadataName** is equal to **DateTime**, it means that the current IdentifierName node represents a **DateTime** type, and therefore it is a node we want to report a diagnostic for.

An important consideration at this point: The Syntax Visualizer tool shows the full list of properties of each syntax node and each symbol, if any. With regard to this example, if you take a look back at Figure 29, you will see how the Syntax Visualizer is showing symbol information for **DateTime**, including the **MetadataName** property. So once again, the Syntax Visualizer tool is very important, because it shows the full list of properties that you can investigate to determine any information about a syntax node.

As another example, the **IMethodSymbol** type exposes a property called **ReturnType**, which contains the fully-qualified name of the type returned by a method, and can be very useful to determine if a method being analyzed is the one you expect. Never forget to keep the Syntax Visualizer open when you want to analyze a syntax node that you do not know very well; it will give you the full list of properties exposed and information returned by any syntax node.

Reporting Diagnostics

Going back to the code, if the value of **MetadataName** is **DateTime**, you need to create a diagnostic that will squiggle the type name in the code editor. To do this, you invoke the static **Create** method from the **Microsoft.CodeAnalysis.Diagnostic** class. This method takes three arguments: the instance of the diagnostic descriptor (**Rule**), the diagnostic position in the code editor (**root.GetLocation**), and a descriptive message. Notice that any syntax node class exposes the **GetLocation** method, which corresponds to the currently investigated syntax node, in this case the **IdentifierNameSyntax** node. Finally, you report the diagnostic by invoking the **ReportDiagnostic** method of the context class. In this way, tools like Visual Studio will receive the diagnostic instance and use its information to highlight the code issue and display information about it.

A Quick Look at Code Fixes

So far, you have completed the first part of the analyzer, which is the diagnostic. The next step is providing a code fix that will help developers solve the code issue by showing potential fixes within the light bulb.

Remember: Offering code fixes is not mandatory. There are plenty of built-in diagnostics in the VB and C# compilers that only report warnings and errors without suggesting code fixes, and there might be many situations and reasons to not provide a code fix. If you think of regular expressions, you might write an analyzer that checks if a string literal does not match a particular regex, but you cannot really predict all the possible ways to write a correct literal. In that case, you might just want to report a code issue.

Tip: Actions you code inside an analyzer are invoked at every keystroke. This means you must design your analyzers efficiently, so that they do not affect the code editor's performance. The secret to creating efficient analyzers is implementing small tests, one at a time, instead of performing one heavy analysis. The preceding example is very efficient, because it implements small tests in sequence before creating and reporting the diagnostic.

You might argue at this point that, for Visual Basic, the sample code is incomplete because this language also offers the **`Date`** keyword, which is used instead of **`DateTime`**. This is a particular scenario of those APIs that are tailored to a specific language, and it is described later in the section Language-Tailored APIs. For now, it's time to provide a code fix.

Implementing Code Fixes

A code fix is a quick action that integrates into the light bulb and suggests a possible solution to a code issue. You implement a code fix in the CodeFixProvider.cs (or .vb) file.

Tip: Always keep in mind that both DiagnosticAnalyzer.cs and CodeFixProvider.cs are conventional names that you can replace with more meaningful file names. For the sake of consistency, I use the default file names in this book.

A code fix is represented by a class that inherits from **`Microsoft.CodeAnalysis.CodeFixProvider`**, which provides common infrastructure for code fixes. For this example, Visual Studio 2015 has generated a class called **`DateTimeAnalyzer_CSCodeFixProvider`**, which inherits from **`CodeFixProvider`** and is decorated with the **`ExportCodeFixProvider`** attribute. The latter requires two arguments: The target language (still specified via the **`LanguageNames`** enumeration) and the name.

`CodeFixProvider` exposes three methods that derived classes must override:

- **`FixableDiagnosticIds`**, which returns an immutable array with one element representing the diagnostic ID associated to it, and can be fixed with the current code fix provider.
- **`GetFixAllProvider`**, which defines a way to provide fixes that you can select from the **`FixAllProvider`** class.
- **`RegisterCodeFixesAsync`**, where you implement the code that solves an issue.

RegisterCodeFixesAsync requires special attention, so the first part of the code for the **DateTimeAnalyzerCodeFixProvider** class is about its definition and about the first two methods in the previous list. Code Listing 7 shows this piece of code.

Code Listing 7 (C#)

```
[ExportCodeFixProvider(LanguageNames.CSharp, Name =
 nameof(DateTimeAnalyzer_CSCodeFixProvider)), Shared]
public class DateTimeAnalyzer_CSCodeFixProvider : CodeFixProvider
{
     private const string title = "Replace with DateTimeOffset";

     public sealed override ImmutableArray<string> FixableDiagnosticIds
     {
         get { return ImmutableArray.
               Create(DateTimeAnalyzerAnalyzer.DiagnosticId); }
     }

     public sealed override FixAllProvider GetFixAllProvider()
     {
         return WellKnownFixAllProviders.BatchFixer;
     }
```

Code Listing 7 (VB)

```
<ExportCodeFixProvider(LanguageNames.VisualBasic,
Name:=NameOf(DateTimeAnalyzer_VBCodeFixProvider)), [Shared]>
Public Class DateTimeAnalyzer_VBCodeFixProvider
    Inherits CodeFixProvider

    Private Const title As String =
            "Replace DateTime with DateTimeOffset"

    Public NotOverridable Overrides ReadOnly Property
           FixableDiagnosticIds As ImmutableArray(Of String)
        Get
            Return ImmutableArray.
                   Create(DateTimeAnalyzer_VBAnalyzer.DiagnosticId)
        End Get
    End Property

    Public NotOverridable Overrides Function _
           GetFixAllProvider() As FixAllProvider
        Return WellKnownFixAllProviders.BatchFixer
    End Function
```

Registering Actions

The core of a code fix is the **RegisterCodeFixesAsync** method, where you register an action that will solve the code issue. Of course, you can provide multiple actions for different potential fixes. Code Listing 8 shows the implementation for the current example (read the code comments for details).

Code Listing 8 (C#)

```
        public sealed override async Task
               RegisterCodeFixesAsync(CodeFixContext context)
        {
            // Get the root syntax node for the current document
            var root = await context.Document.
                GetSyntaxRootAsync(context.CancellationToken).
                ConfigureAwait(false);

            // Get a reference to the diagnostic to fix
            var diagnostic = context.Diagnostics.First();
            // Get the location in the code editor for the diagnostic
            var diagnosticSpan = diagnostic.Location.SourceSpan;

            // Find the syntax node on the span
            // where there is a squiggle
            var node = root.FindNode(context.Span);

            // If the syntax node is not an IdentifierName
            // return
            if (node is IdentifierNameSyntax == false)
            {
                return;
            }

            // Register a code action that invokes the fix
            // on the current document
            context.RegisterCodeFix(
            CodeAction.Create(title:title,
                              createChangedDocument:
                              c=> ReplaceDateTimeAsync(context.Document,
                              node, c),
                              equivalenceKey:title), diagnostic);
        }
```

Code Listing 8 (VB)

```
    Public NotOverridable Overrides Async Function _
        RegisterCodeFixesAsync(context As CodeFixContext) As Task

        'Get the root syntax node for the current document
```

```
        Dim root = Await context.Document.
            GetSyntaxRootAsync(context.CancellationToken).
            ConfigureAwait(False)

        'Get a reference to the diagnostic to fix
        Dim diagnostic = context.Diagnostics.First()

        'Get the location in the code
        'editor for the diagnostic
        Dim diagnosticSpan =
            diagnostic.Location.SourceSpan
        'Find the syntax node on the span
        'where there is a squiggle
        Dim node = root.FindNode(context.Span)

        'Register a code action that invokes the fix
        'on the current document
        context.RegisterCodeFix(
        CodeAction.Create("Replace with DateTimeOffset",
                          Function(c)
                          ReplaceDateTimeAsync(context.Document,
                          node, c), equivalenceKey:=title),
                          diagnostic)
    End Function
```

The method takes a parameter called **context**, of type **Microsoft.CodeAnalysis.CodeFixes.CodeFixContext**, which is a value type that represents the current context for code fixes. Context refers to syntax nodes that have diagnostics.

Getting a Reference to Diagnostics

The **RegisterCodeFixesAsync** method first retrieves the root syntax node for the current document (that is, the source code file) via the **GetSyntaxRootAsync** method. Next, it gets a reference to the diagnostic to fix by invoking **First** over the immutable array of code fixes. After that, the code retrieves the diagnostic position in the code editor via the **diagnostic.Location.SourceSpan** property and retrieves the related syntax node via the **root.FindNode** method.

Integrating Registered Actions with the Light Bulb

The last step in this method is registering a code action via the static **Microsoft.CodeAnalysis.CodeAction.Create** method, which takes as parameters the title that will be shown in the light bulb, a delegate that performs the actual work to fix the syntax node, a unique identifier called **equivalenceKey**, and the instance of the diagnostic. This delegate, which is called **ReplaceDateTimeAsync** in the current example, typically receives three arguments: An instance of the **Document** class representing the source code file that contains the syntax node you want to fix, the instance of the syntax node, and a cancellation token.

Generating a New Syntax Node

At this point, the goal is creating a new syntax node where the code issue has been solved based on the syntax node that has the diagnostic, and replacing the latter with the new node, returning a new **`Document`** instance. This is demonstrated in Code Listing 9.

Code Listing 9 (C#)

```
        private async Task<Document>
                ReplaceDateTimeAsync(Document document,
                        SyntaxNode node,
                        CancellationToken cancellationToken)
        {
            // Get the root syntax node for the current document
            var root = await document.GetSyntaxRootAsync();

            // Convert the syntax node into the specialized kind
            var convertedNode = (IdentifierNameSyntax)node;

            // Create a new syntax node
            var newNode = convertedNode?.WithIdentifier(SyntaxFactory.
                          ParseToken("DateTimeOffset")).
                          WithLeadingTrivia(node.GetLeadingTrivia()).
                          WithTrailingTrivia(node.GetTrailingTrivia());

            // Create a new root syntax node for the current document,
            // replacing the syntax node that has the diagnostic with
            // a new syntax node
            var newRoot = root.ReplaceNode(node, newNode);

            // Generate a new document
            var newDocument = document.WithSyntaxRoot(newRoot);
            return newDocument;
        }
```

Code Listing 9 (VB)

```
    Private Async Function ReplaceDateTimeAsync _
            (document As Document,
             node As SyntaxNode,
             cancellationToken As CancellationToken) _
             As Task(Of Document)

        'Get the root syntax node for the current document
        Dim root = Await document.GetSyntaxRootAsync

        'Convert the syntax node into the specialized kind
        Dim convertedNode = DirectCast(node, IdentifierNameSyntax)
```

```
        'Create a new syntax node
        Dim newNode = convertedNode.
                      WithIdentifier(SyntaxFactory.
                      ParseToken("DateTimeOffset")).
                      WithTrailingTrivia(node.
                      GetTrailingTrivia)

        'Create a new root syntax node for the current document,
        'replacing the syntax node that has the diagnostic with
        'a new syntax node
        Dim newRoot = root.ReplaceNode(node, newNode)

        'Generate a new document
        Dim newDocument = document.WithSyntaxRoot(newRoot)
        Return newDocument

    End Function
```

The first two lines of code retrieve the root syntax node for the current document and convert the current syntax node instance into the expected type, which is **`IdentifierNameSyntax`**. The core of the logic is in the **`newNode`** variable declaration; the goal is modifying the syntax node that has diagnostics with the **`DateTimeOffset`** type name instead of **`DateTime`**. Syntax nodes are immutable, so you cannot edit **`convertedNode`** directly. Instead, you create a new syntax node (**`newNode`**) starting from the existing one.

To make your edit, you invoke the **`WithIdentifier`** method, which allows supplying a new identifier to an **`IdentifierNameSyntax`** node. **`WithIdentifier`**'s argument is a **`SyntaxToken`**; because you cannot supply a **`SyntaxNode`** directly, you use the **`ParseToken`** method from the **`SyntaxFactory`** class, which parses a string and returns a **`SyntaxToken`**. Because the original syntax node might have white spaces, comments, and terminators, you also invoke the **`WithTrailingTrivia`** method. Since you don't need to change the original trivia, you simply pass the invocation to **`node.GetTrailingTrivia`**, which obtains the trailing trivia from the original syntax node.

The next step is generating a new root syntax node. Because syntax nodes are immutable, you cannot edit the original root syntax node directly. Instead, you create a new one by replacing the old syntax node with the new node (**`ReplaceNode`**).

Finally, you generate a new document by invoking **`WithSyntaxRoot`** and passing the new root syntax node as the argument. This invocation returns a new **`Document`** instance that represents the current source file where the code issue has been fixed. Before doing any other work, it is important to stop for a moment and discuss important objects in the Roslyn APIs.

Tip: Types deriving from `SyntaxNode` expose a number of `With` methods, which you invoke to create a new syntax node, supplying updated information. Using `With` methods is very common when implementing code fixes and refactorings, and IntelliSense is your

best friend in this, showing the full list of available methods and their descriptions for each class that inherits from `SyntaxNode`.

Generating Syntax Nodes: SyntaxFactory and SyntaxGenerator

With Roslyn, you can perform different kinds of code-related tasks, such as code generation and analysis, starting from source text or working against existing syntax nodes. In the second case, because of immutability, you create a new syntax node based on an existing one and make the proper edits, which includes adding or removing members.

Whether you need to create new syntax trees and nodes from scratch, or make edits to an existing syntax node, you need to work with syntax nodes. The .NET Compiler Platform offers two objects for generating any kind of syntax nodes: the **`SyntaxFactory`** class and the **`SyntaxGenerator`** class. Let's start with **`SyntaxFactory`**.

Generating Syntax Nodes with SyntaxFactory

There are specific versions of this class for Visual Basic (**`Microsoft.CodeAnalysis.VisualBasic.SyntaxFactory`**) and C# (**`Microsoft.CodeAnalysis.CSharp.SyntaxFactory`**) because these languages have different lexicons and syntax. For instance, in C# you have brackets to delimit a method definition, whereas in Visual Basic you have the **`Sub`** and **`Function`** keywords, plus an **`End`** statement.

Because every single thing you type must have a syntactic representation, you can understand why there are two different implementations of **`SyntaxFactory`**. This exposes a number of methods that allow generating syntax elements. For instance, imagine you want to create a syntax node that represents the class shown in Code Listing 10.

Code Listing 10 (C#)

```
public abstract class Person : IDisposable
{
    public void IDisposable.Dispose()
    {
        // Leaving blank for simplicity
    }
}
```

Code Listing 10 (VB)

```
Public MustInherit Class Person
    Implements IDisposable

    Public Sub Dispose() Implements IDisposable.Dispose
        'Leaving blank for simplicity
    End Sub
End Class
```

Though this class definition might seem very simple, it has enough syntax members for you to understand how the syntax generation works. In fact, here you have a class definition with modifiers and accessors. The class implements an interface, and exposes a method that implements a method required by the interface.

Take a moment to browse the definition of this simple code with the Syntax Visualizer. Once you have a more precise idea of the syntax elements that compose this node, look at Code Listing 11 and discover how you can generate the syntactic definition with **`SyntaxFactory`**.

Code Listing 11 (C#)

```
            var classBlock = SyntaxFactory.ClassDeclaration(
            @"Person")
        .WithModifiers(
            SyntaxFactory.TokenList(
                new[]{
                    SyntaxFactory.Token(
                        SyntaxKind.AbstractKeyword),
                    SyntaxFactory.Token(
                        SyntaxKind.PublicKeyword)}))
        .WithKeyword(
            SyntaxFactory.Token(
                SyntaxKind.ClassKeyword))
        .WithBaseList(
            SyntaxFactory.BaseList(
                SyntaxFactory.SingletonSeparatedList<BaseTypeSyntax>(
                    SyntaxFactory.SimpleBaseType(
                        SyntaxFactory.IdentifierName(
                            @"IDisposable"))))
            .WithColonToken(
                SyntaxFactory.Token(
                    SyntaxKind.ColonToken)))
        .WithOpenBraceToken(
            SyntaxFactory.Token(
                SyntaxKind.OpenBraceToken))
        .WithMembers(
            SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
                SyntaxFactory.MethodDeclaration(
                    SyntaxFactory.PredefinedType(
                        SyntaxFactory.Token(
                            SyntaxKind.VoidKeyword)),
                    SyntaxFactory.Identifier(
                        @"Dispose"))
                .WithModifiers(
                    SyntaxFactory.TokenList(
                        SyntaxFactory.Token(
                            SyntaxKind.PublicKeyword)))
                .WithExplicitInterfaceSpecifier(
                    SyntaxFactory.ExplicitInterfaceSpecifier(
```

```
                        SyntaxFactory.IdentifierName(
                            @"IDisposable"))
                    .WithDotToken(
                        SyntaxFactory.Token(
                            SyntaxKind.DotToken)))
                .WithParameterList(
                    SyntaxFactory.ParameterList()
                    .WithOpenParenToken(
                        SyntaxFactory.Token(
                            SyntaxKind.OpenParenToken))
                    .WithCloseParenToken(
                        SyntaxFactory.Token(
                            SyntaxKind.CloseParenToken)))
                .WithBody(
                    SyntaxFactory.Block()
                    .WithOpenBraceToken(
                        SyntaxFactory.Token(
                            SyntaxKind.OpenBraceToken))
                    .WithCloseBraceToken(
                        SyntaxFactory.Token(
                            SyntaxKind.CloseBraceToken)))))
        .WithCloseBraceToken(
            SyntaxFactory.Token(
                SyntaxKind.CloseBraceToken)).
                NormalizeWhitespace();
```

Code Listing 11 (VB)

```
Dim ClassBlock = SyntaxFactory.ClassBlock(
                 SyntaxFactory.ClassStatement("Person").
                 AddModifiers(SyntaxFactory.ParseToken("MustInherit"),
                              SyntaxFactory.ParseToken("Public"))).
                 AddImplements(SyntaxFactory.
                 ImplementsStatement(SyntaxFactory.
                 ParseTypeName("IDisposable"))).
                 AddMembers(SyntaxFactory.
                 MethodBlock(SyntaxKind.SubBlock,
                             SyntaxFactory.SubStatement(Nothing,
                             SyntaxFactory.TokenList(
                             SyntaxFactory.ParseToken("Public")),
                             SyntaxFactory.ParseToken("Dispose()"),
                             Nothing, Nothing, Nothing, Nothing,
                             SyntaxFactory.ImplementsClause(
                             SyntaxFactory.ParseToken("Implements"),
                             SyntaxFactory.
                         SingletonSeparatedList(Of QualifiedNameSyntax) _
                             (SyntaxFactory.QualifiedName(SyntaxFactory.
                             ParseName("IDisposable"),
```

```
                                SyntaxFactory.ParseName("Dispose")))),
                                SyntaxFactory.EndSubStatement)).
                NormalizeWhitespace()
```

Creating Syntax Nodes Programmatically

Each syntax node you see in the Syntax Visualizer tool has a corresponding method exposed by the **SyntaxFactory** class. These methods allow for creating syntax nodes, and their names are really self-explanatory; the arguments they take are other syntax nodes or syntax tokens.

For instance, the **ClassBlock** method in Visual Basic generates a **Class..End Class** block and takes an argument which is the **ClassStatement** method. This **ClassStatement** method represents the **Class** statement (which includes the class name), modifiers (such as **Public** and **MustInherit** in Code Listing 11), **Implements** statements (**AddImplement** plus the **ImplementsStatements**), **Inherits** statements, and so on.

Adding Members to Syntax Nodes

The **ClassBlock** method also allows adding members, such as methods, to the class, each represented by an invocation to the **MethodBlock** method. It is worth mentioning the **SyntaxFactory.QualifiedName**, which generates a fully-qualified type name, and **SyntaxFactory.ParseName**, which parses a type name and returns an **IdentifierNameSyntax** object.

The same considerations apply to C#, but method names are slightly different, according to the different lexicon and semantics. It is always important to invoke the **NormalizeWhitespace** method, which supplies the proper indentation and line terminators to the generated syntax nodes.

Using the **SyntaxFactory** class with the help of IntelliSense is so straightforward that listing any possible method is not necessary at this point, and once you understand how any syntax nodes you see in the Syntax Visualizer can be created with methods exposed by **SyntaxFactory**, everything will be much easier.

Generating Syntax Nodes with SyntaxGenerator

Though the **SyntaxFactory** class is very powerful and allows generating any possible syntax element, you use it differently according to the programming language you are working with. The .NET Compiler Platform also offers the **Microsoft.CodeAnalysis.Editing.SyntaxGenerator** class, which is a language-agnostic factory for creating syntax nodes. This class allows for creating language-specific syntax nodes that are semantically similar between languages. In simpler words, it exposes the same members for both VB and C#, and you use it the same way against both languages.

In order to use **SyntaxGenerator**, you need an instance of this class that you get by invoking its static **GetGenerator** method. This requires either a **Document** instance (which represents a source code file) or a workspace. In the case of a **Document**, you can work with syntax nodes within an existing code file and **GetGenerator** will automatically detect the language it's written with. In the case of a workspace, you can also create syntax nodes from scratch, and you pass

both an instance of the **`MSBuildWorkspace`** class and the name of the programming language as arguments.

The Workspaces APIs and the **`MSBuildWorkspace`** class will be discussed in Chapter 8, but what you need to know at this point is simply that this type represents a workspace that can be populated with MSBuild solutions and projects. Code Listing 12 shows how to generate a syntax node that represents the content of Code Listing 10, using **`SyntaxGenerator`**. You will be surprised at how both C# and Visual Basic use exactly the same APIs.

Note: In order to complete and compile the example shown in Code Listing 12, you can either create a console application and then install the *Microsoft.CodeAnalysis* ***package from NuGet, or you can create a console application using the*** *Stand-Alone Code Analysis Tool* ***project template, which is available in the*** *Extensibility* ***node of the*** *New Project* ***dialog. The latter is discussed more thoroughly in Chapter 8, and automatically installs all the necessary NuGet packages to work with Roslyn.***

Code Listing 12 (C#)

```
//Requires the following using directives:
//using Microsoft.CodeAnalysis;
//using Microsoft.CodeAnalysis.Editing;
//using Microsoft.CodeAnalysis.MSBuild;
    static void Main(string[] args)
    {
        //Create a workspace
        var ws = MSBuildWorkspace.Create();

        //Get an instance of SyntaxGenerator for
        //the specified workspace in C#
        var generator = SyntaxGenerator.
                        GetGenerator(ws, "C#");

        //Generate a qualified name for IDisposable
        var interfaceType =
            generator.DottedName("IDisposable");

        //Generate a public method called Dispose
        var methodBlock =
            generator.MethodDeclaration(
                      "Dispose",
                      accessibility:
                      Accessibility.Public);

        //Make the method implement IDisposable.Dispose
        var methodBlockWithInterface =
            generator.
            AsPublicInterfaceImplementation(methodBlock,
            interfaceType);
```

```
            //Generate a public abstract class
            //that implements the previous interface
            //and with the specified members
            var classBlock =
                generator.ClassDeclaration("Person",
                          accessibility:
                          Accessibility.Public,
                          modifiers:
                          DeclarationModifiers.Abstract,
                          members:new SyntaxNode[]
                          { methodBlockWithInterface },
                          interfaceTypes:new SyntaxNode[]
                          { interfaceType }).
                          NormalizeWhitespace();

            Console.WriteLine(classBlock.ToFullString());
            Console.Read();
        }
```

Code Listing 12 (VB)

```
    'Requires the following Imports directives:
    'Imports Microsoft.CodeAnalysis
    'Imports Microsoft.CodeAnalysis.Editing
    'Imports Microsoft.CodeAnalysis.MSBuild
    Sub Main()
        'Create a workspace
        Dim ws = MSBuildWorkspace.Create()

        'Get an instance of SyntaxGenerator for
        'the specified workspace in VB
        Dim generator = SyntaxGenerator.
                        GetGenerator(ws,
                        "Visual Basic")

        'Generate a qualified name for IDisposable
        Dim interfaceType =
            generator.DottedName("IDisposable")

        'Generate a Public method called Dispose
        Dim methodBlock =
            generator.MethodDeclaration(
                      "Dispose",
                      accessibility:=
                      Accessibility.Public)

        'Make the method implement IDisposable.Dispose
        Dim methodBlockWithInterface =
```

```
            generator.
            AsPublicInterfaceImplementation(methodBlock,
            interfaceType)

        'Generate a public abstract class
        'that implements the previous interface
        'and with the specified members
        Dim classBlock =
            generator.ClassDeclaration("Person",
                      accessibility:=
                      Accessibility.Public,
                      modifiers:=
                      DeclarationModifiers.Abstract,
                      members:={methodBlockWithInterface},
                      interfaceTypes:={interfaceType}).
                      NormalizeWhitespace

        Console.WriteLine(classBlock.ToFullString)
        Console.Read()
    End Sub
```

Here are a few things to note at this point:

- You use the same APIs for both Visual Basic and C#.
- **SyntaxGenerator** allows generating syntax nodes via methods that represent the semantics of both Visual Basic and C#. By using IntelliSense, you will see the full list of methods, whose names recall the name of the type or member they generate (e.g., **ClassDeclaration** to generate a class, **InterfaceDeclaration** to generate an interface, **PropertyDeclaration** to generate a property, and so on).
- Methods from **SyntaxGenerator** return **SyntaxNode** and, in many cases, their parameters are either of type **SyntaxNode**, or collections or arrays of **SyntaxNode** objects. Working with **SyntaxNode** instead of derived objects gives **SyntaxGenerator** great flexibility.
- You can invoke the **DottedName** method to parse a qualified type name into a **SyntaxNode** that can be passed as an argument to any method that requires a **SyntaxNode** for representing a type name.

As for **SyntaxFactory**, both IntelliSense and the source definition will help you get the full list of available methods and understand their purpose.

Note: You might wonder why you should use** SyntaxFactory **when** SyntaxGenerator **provides a language-agnostic way to generate syntax nodes. First,** SyntaxFactory **fits well with syntax generation with APIs that are tailored for a specific language. Second,** SyntaxFactory **has existed from the very first community technology previews of Roslyn, and has been widely used over the years to create syntax nodes.** SyntaxGenerator **is much more recent, and does not have the same diffusion yet. If you search for examples about generating syntax nodes on the web, you will find that more than 90 percent of the examples use** SyntaxFactory**. For these reasons, you must have knowledge of both types.

Language-Tailored APIs

Though the Diagnostic APIs offer a huge number of types and members that are common to both C# and Visual Basic, there are situations where you need APIs that are tailored for a given language. With regard to the sample analyzer discussed so far, the Visual Basic language has the **Date** keyword, which you can use instead of **System.DateTime** (and which has the same return type). For the sake of completeness, you have to perform code analysis over the **Date** keyword as well. With the help of the Syntax Visualizer, you can discover how the **Date** keyword is represented by a syntax node called PredefinedType, mapped by a **PredefinedTypeSyntax** object. This implies that you need to register an additional action in the **Initialize** method, and you need to perform code analysis over this kind of node. That said, you have to perform the following tasks:

- Register an additional syntax node action that checks for the PredefinedType syntax kind, in the **Initialize** method of the diagnostic class.
- Edit the **AnalyzeDateTime** method in the diagnostic class to check if the current syntax node is either an IdentifierName or a PredefinedType.
- In the code fix provider class, register an additional action that will also offer a fix for the PredefinedType syntax node.
- Implement a new method that specifically works against the **Date** keyword to solve the code issue.

Code Listing 13 shows edits to the DiagnosticAnalyzer.vb file, whereas Code Listing 14 shows edits in the CodeFixProvider.vb file.

Code Listing 13 (VB)

```
    Public Overrides Sub Initialize(context As AnalysisContext)
        context.RegisterSyntaxNodeAction(AddressOf AnalyzeDateTime,
                                         SyntaxKind.PredefinedType)
        context.RegisterSyntaxNodeAction(AddressOf AnalyzeDateTime,
                                         SyntaxKind.IdentifierName)
    End Sub

    Private Sub AnalyzeDateTime(context As SyntaxNodeAnalysisContext)

        'Get the syntax node to analyze
        Dim root = context.Node

        'If it's not an IdentifierName syntax node,
        'return
        If TypeOf (root) Is PredefinedTypeSyntax Then
            root = CType(context.Node, PredefinedTypeSyntax)
        ElseIf TypeOf (root) Is IdentifierNameSyntax
            'Conver to IdentifierNameSyntax
            root = CType(context.Node, IdentifierNameSyntax)
        Else
            Return
        End If
```

```
        'Get the symbol info for
        'the DateTime type declaration
        Dim dateSymbol =
            TryCast(context.SemanticModel.
                    GetSymbolInfo(root).Symbol,
                    INamedTypeSymbol)

        'If no symbol info, return
        If dateSymbol Is Nothing Then
            Return
        End If

        'If the name  of the symbol is not
        'DateTime, return
        If Not dateSymbol.MetadataName = "DateTime" Then
            Return
        End If

        'Create a diagnostic at the node location
        'with the specified message and rule info
        Dim diagn =
            Diagnostic.Create(Rule, root.GetLocation,
                        "Consider replacing with DateTimeOffset")

        'Report the diagnostic
        context.ReportDiagnostic(diagn)
    End Sub
```

Code Listing 14 (VB)

```
    Private Async Function ReplaceDateTimeAsync _
            (document As Document,
             node As SyntaxNode,
             cancellationToken As CancellationToken) _
             As Task(Of Document)

        'Get the root syntax node for the current document
        Dim root = Await document.GetSyntaxRootAsync

        Dim newRoot As SyntaxNode = Nothing

        'Convert the syntax node into the specialized kind
        If TypeOf (node) Is IdentifierNameSyntax Then
            Dim convertedNode = DirectCast(node, IdentifierNameSyntax)

            'Create a new syntax node
            Dim newNode = convertedNode.
```

```
                        WithIdentifier(SyntaxFactory.
                        ParseToken("DateTimeOffset")).
                        WithTrailingTrivia(node.
                        GetTrailingTrivia)

            newRoot = root.ReplaceNode(node, newNode)
        ElseIf TypeOf (node) Is PredefinedTypeSyntax
            Dim convertedNode = DirectCast(node, PredefinedTypeSyntax)

            Dim newIdentifierName = SyntaxFactory.
                        IdentifierName(SyntaxFactory.
                        ParseToken("DateTimeOffset")).
                        WithTrailingTrivia(node.
                        GetTrailingTrivia)
            newRoot = root.ReplaceNode(node, newIdentifierName)
        End If

        'Create a new root syntax node for the current document,
        'replacing the syntax node that has the diagnostic with
        'a new syntax node

        'Generate a new document
        Dim newDocument = document.WithSyntaxRoot(newRoot)
        Return newDocument
    End Function
```

As you can see in Code Listing 13, you need an additional check to determine if the syntax node is either a **PredefinedTypeSyntax** or an **IdentifierNameSyntax**, and perform the proper conversion. You do not need to change the rest of the code because the symbol information for **Date** is the still obtained from the same **SyntaxNode**.

In Code Listing 14, you see how the **ReplaceDateTimeAsync** method has been enhanced to check if the node is either an **IdentifierNameSyntax** or a **PredefinedTypeSyntax**, and to supply a new syntax node based on the node's type. In the current example, you are not changing properties of an existing node (PredefinedType). Instead, you need to create a totally different kind of syntax node (IdentifierName) that will replace the previous one in the root node, so you cannot use any methods whose names begin with **With**.

Here you create a new **IdentifierNameSyntax**, using the **SyntaxFactory.IdentifierName** method, which returns an object of type **IdentifierNameSyntax** and takes an argument of type **SyntaxToken**. This syntax token contains the name of the identifier (**DateTimeOffset**) and is generated with **SyntaxFactory.ParseToken**. The newly generated **IdentifierNameSyntax** will replace the **PredefinedTypeSyntax** in the root syntax node. With this approach, you have also addressed the requirement of providing a code fix for the **Date** keyword as well. This will be demonstrated in practice shortly, when testing the analyzer.

Hints for Improving the Sample Analyzer

The DateTimeAnalyzer sample has been designed to make it easier for you to understand the concepts and the logic behind the Diagnostic APIs. However, it has some limitations, and it can be certainly improved in different ways. For example, because the constructor of **DateTimeOffset** has an overload that takes one argument of type **DateTime**, when the code declares a new instance of a **DateTime** object, you should consider providing a code fix that specifies the **DateTime** declaration as the argument of the **DateTimeOffset**'s constructor like this: **DateTimeOffset myDate = new DateTimeOffset(new DateTime(2015, 09, 09));**.

Additionally, for Windows Store and Windows Phone apps, you could restrict the diagnostic to be reported only if a **DateTime** is assigned to the **Date** property of the **DatePicker** control (which is of type **DateTimeOffset**). In this way, the improper assignment to the user control would be reported as a code issue, but developers would be free to use **DateTime** outside of that context. With the help of the Syntax Visualizer, it will be easier to understand which syntax nodes you need to work with for these scenarios.

Restricting Code Analysis to Specific Platforms

In many cases, you might want diagnostics to be applied only in certain development platforms. For instance, the diagnostic in the current sample analyzer makes sense only within Windows Store, Windows Phone, and OData Web API applications. The **Compilation** class exposes a method called **GetTypeByMetadataName** that can be used to determine if the analyzed project has a reference to any libraries that define the specified type name. If there is no reference, the method returns null; a null result means that your analyzer is not running on a given platform.

For a better understanding, let's consider the case of Windows Store and Windows Phone apps. The Windows Runtime defines an object called **Windows.Storage.StorageFile**, which represents a file on disk. You can invoke **Compilation.GetTypeByMetadataName** by passing **Windows.Storage.StorageFile** or any other type name defined in the Windows Runtime, to detect if the project has a reference to libraries that define this type. If the result is null, it means that your analyzer is running on neither Windows 8.1 nor Windows Phone 8.1, and therefore, the code analysis can be skipped (which also improves performance).

Many developers used to place this check inside the delegate that performs the code analysis (**AnalyzeDateTime** in the current example). This definitely works, but the check would be executed every time the delegate is invoked (that is, at every key stroke). A better approach is to register a compilation start action in the **Initialize** method and perform this check only once. To accomplish this, you invoke the **AnalysisContext.RegisterCompilationStartAction**, which executes an action when compilation starts. Code Listing 15 demonstrates this.

Code Listing 15 (C#)

```
    public override void Initialize(AnalysisContext context)
    {
        context.
        RegisterCompilationStartAction(
```

```
        (CompilationStartAnalysisContext ctx) =>
        {
            var requestedType =
                ctx.Compilation.
                GetTypeByMetadataName("Windows.Storage.StorageFile");

            if (requestedType == null)
                return;

            ctx.RegisterSyntaxNodeAction(AnalyzeDateTime,
                SyntaxKind.IdentifierName);
        });
    }
```

Code Listing 15 (VB)

```
    Public Overrides Sub Initialize(context As AnalysisContext)
        context.RegisterCompilationStartAction(
            Sub(ctx As CompilationStartAnalysisContext)
                Dim requestedType =
                    ctx.Compilation.
                    GetTypeByMetadataName("Windows.Storage.StorageFile")
                If requestedType Is Nothing Then Return

                ctx.RegisterSyntaxNodeAction(AddressOf AnalyzeDateTime,
                                             SyntaxKind.PredefinedType)
                ctx.RegisterSyntaxNodeAction(AddressOf AnalyzeDateTime,
                                             SyntaxKind.IdentifierName)
            End Sub)

    End Sub
```

If **`GetTypeByMetadataName`** returns null, it means there is no reference to **`Windows.Storage.StorageFile`**, and therefore the analyzer is not running on Windows 8.1 or Windows Phone 8.1 projects. If it returns a non-null result, the syntax node action is finally registered.

Testing and Debugging an Analyzer

It's time to test and debug the sample analyzer. Generally speaking, to test an analyzer, you first make sure the .Vsix project is set as the startup project, and then you just press F5. At this point, Visual Studio 2015 deploys and installs the auto-generated VSIX package for the analyzer to the experimental instance of the IDE. When this starts, you can simply create a new project or open an existing one and test if your analyzers work.

For the current sample analyzer, make sure the DateTimeAnalyzer.Vsix project is the startup project, and then press F5. When the experimental instance is ready, create a new Windows 8.1 project (or a Windows Phone 8.1 project) and place a **`DateTime`** declaration anywhere in the code editor (see Figure 31).

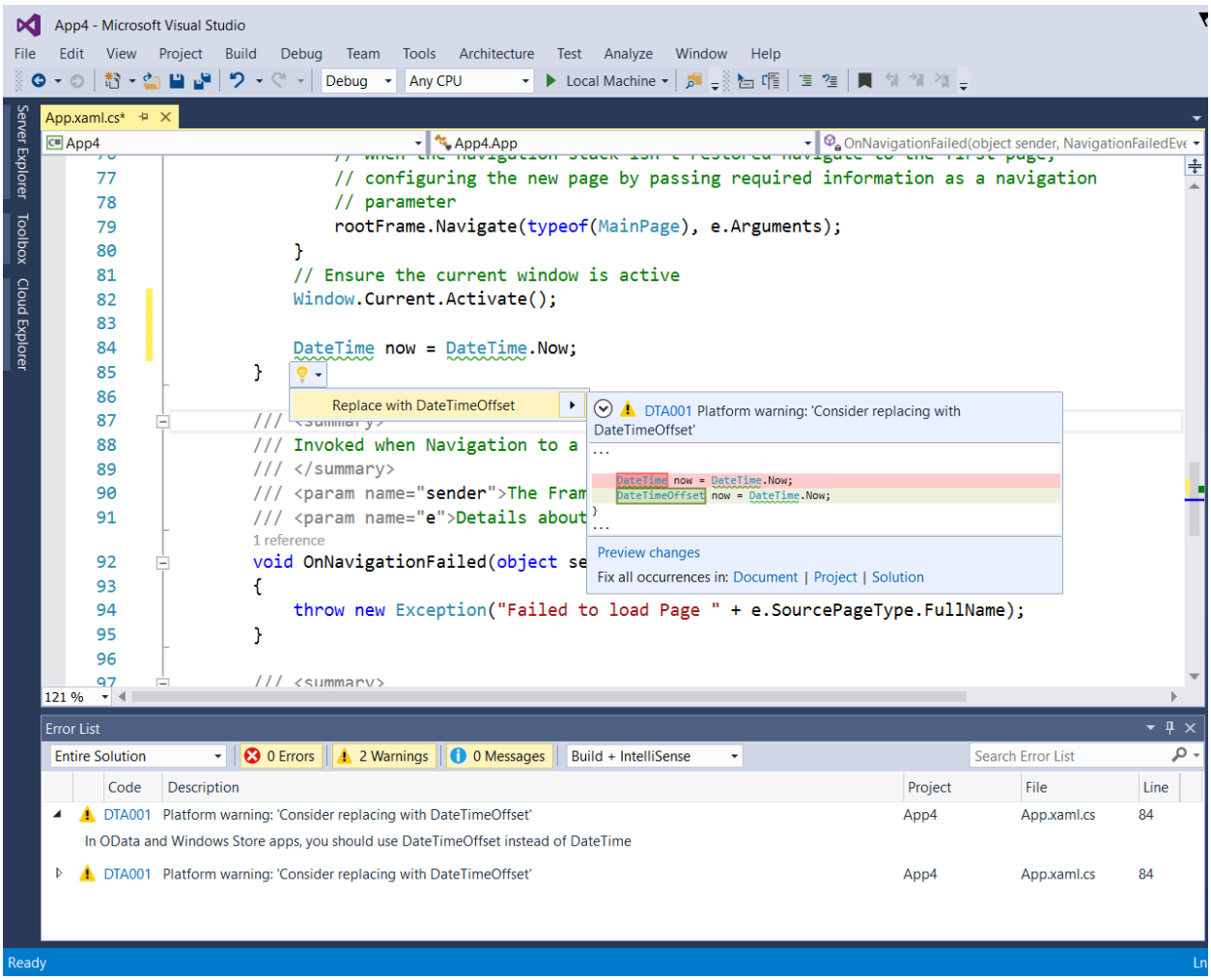

Figure 31: The DateTimeAnalyzer detects the expected issue and suggests code fixes

As shown in Figure 31, the use of **`DateTime`** is correctly identified as a code issue, so the code editor reports a warning with the specified message and diagnostic ID. The light bulb suggests fixes and shows the live preview. You can decide to apply the code fix to all the instances in the document or in the solution. Also notice how the Error List window shows the full message for the code issue, and if you click the error code, you will be able to visit the documentation page on GitHub.

For the sake of completeness, you could also try to create a different project and see how the analyzer will not work because it has been restricted to specific development platforms. You can also use the full debugging instrumentation offered by Visual Studio 2015 to debug an analyzer. This will be important when an analyzer does not work as expected. You can place breakpoints and use all the diagnostic windows you already know, as demonstrated in Figure 32.

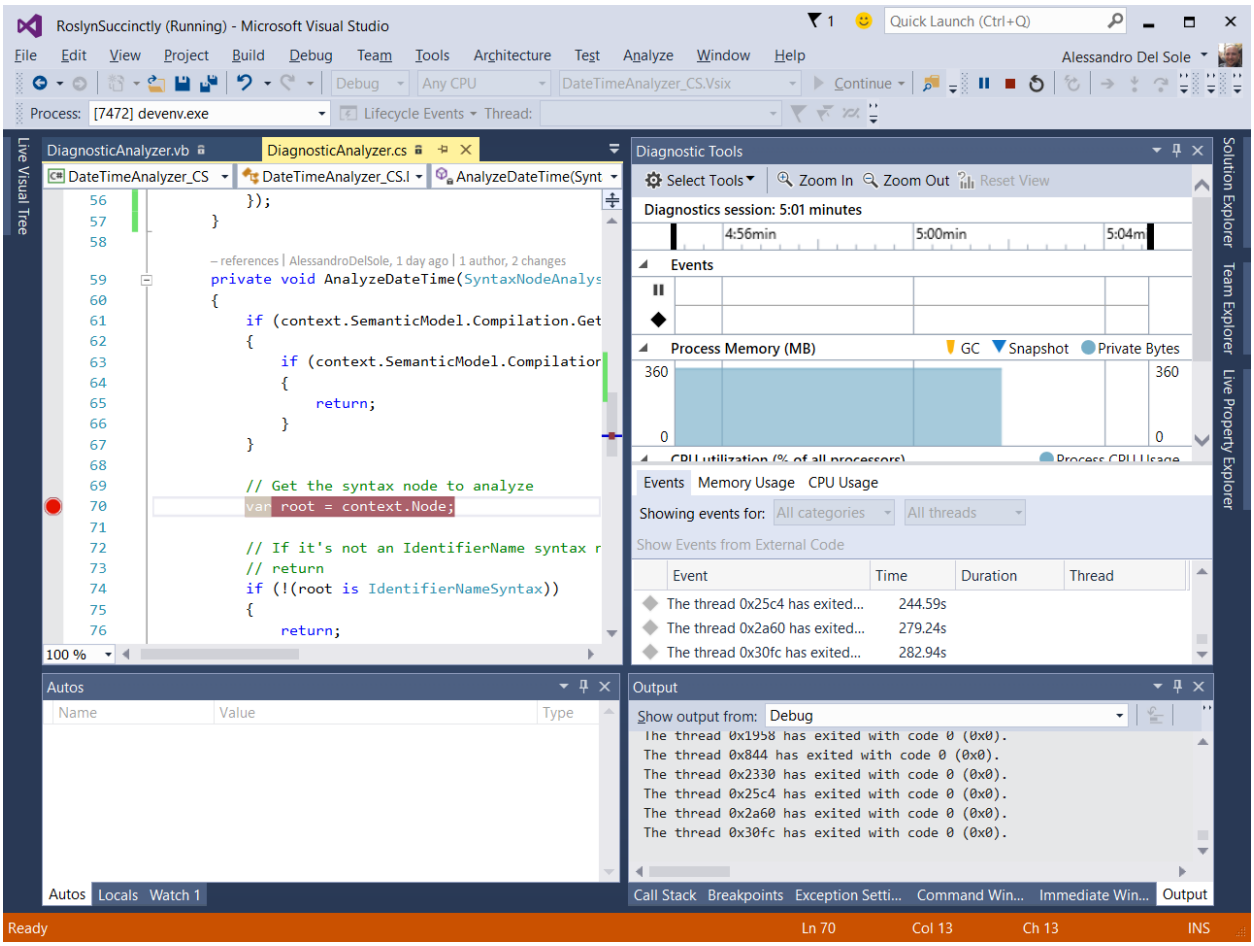

Figure 32: Debugging an analyzer

You can manage installed analyzers and code refactorings in the experimental instance by selecting **Tools** > **Extensions and Updates**. Figure 33 shows how the sample analyzer appears inside the list of installed extensions.

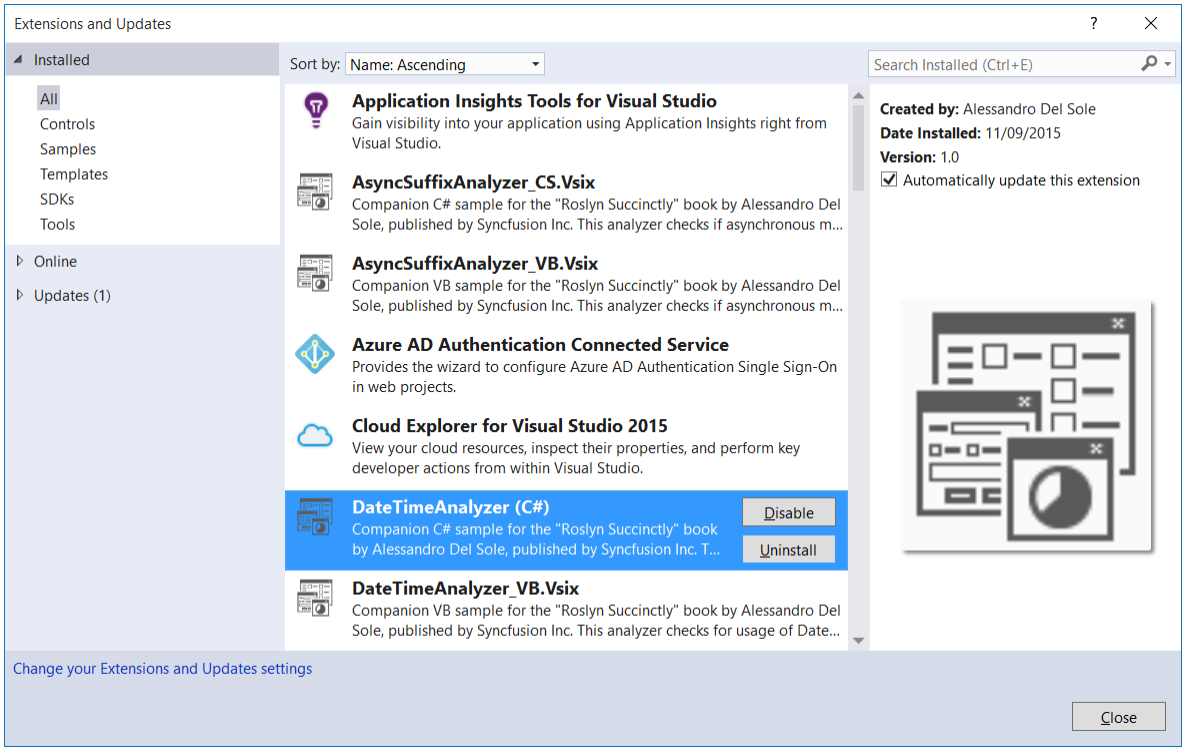

Figure 33: Managing installed analyzers and refactorings in the experimental instance

You can disable or remove installed analyzers as you would do with any other Visual Studio extension. This gives you an idea of what happens when an analyzer or code refactoring is distributed to the public as a VSIX package; this is an important topic and is described in Chapter 7, "Deploying Analyzers and Refactorings to the Visual Studio Gallery".

Tip: If you uninstall an analyzer from the experimental instance, the next time you open the analyzer project in Visual Studio and press F5 for debugging, the build process could fail. This is because the analyzer is not found in the experimental instance. Should this happen, upgrade the VSIX version number in the VSIX manifest designer that you enable by double-clicking the** *Source.vsixmanifest* ***file in the *Solution Explorer*. ***At this point, rebuilding the solution will succeed.***

Adding Multiple Diagnostics to an Analyzer Library

An analyzer library you create via the Diagnostic with Code Fix project template can contain and expose many diagnostics and code fixes. The .NET Compiler Platform SDK also installs the following specific item templates to Visual Studio:

- **Analyzer**: This adds a new code file containing a diagnostic analyzer to the project.
- **Code Fix**: This adds a new code file containing a code fix to the project and requires the project to contain at least a diagnostic. Remember to edit the auto-generated code in order to match the proper diagnostic ID.
- **Refactoring**: This adds a new code file containing a code refactoring to the project (see Chapter 5).

To use one of these templates, either select **Project** > **Add New Item** or right-click the project name in **Solution Explorer**, then select **Add** > **New Item**. When the Add New Item dialog appears, select the **Extensibility** node.

Chapter Summary

In this chapter, you have built your first Roslyn code analyzer. You learned how to use the Syntax Visualizer to understand the syntax elements you need to work with, and then you started working with the Diagnostic with Code Fix project template. You saw the most important .NET objects in a code analyzer, and you wrote a diagnostic, learning where and how to implement the code analysis logic with a number of syntactical elements that you will often find when working with the Roslyn APIs. Next, you implemented a code fix that integrates into the light bulb and fixes a code issue. You discovered important concepts about generating syntax elements, APIs tailored for the VB and C# languages, and how to restrict code analysis to specific development platforms.

Finally, you learned about testing and debugging an analyzer, discovering how you can use the debugging tools in Visual Studio 2015 that you already know. You are at a very good point now—you have in your hands all you need to create great diagnostics. But to complete the job, you have to learn how to create a code refactoring, which is the topic of the next chapter.

Chapter 5 Writing Refactorings

Refactoring code means reorganizing portions of code a better way, without changing the original behavior. Visual Studio 2015 dramatically enhances the refactoring experience, improving the tooling in C# and introducing support for refactoring to Visual Basic (VB) for the first time. The code refactoring tooling is built upon the .NET Compiler Platform, and as is the case for code analyzers, you can write your own domain-specific refactorings that integrate into light bulbs. In this chapter, you'll learn how to create a custom refactoring and revisit many techniques you learned in Chapter 4. Remember: the Syntax Visualizer window is always your best friend.

Creating a Code Refactoring

Code refactorings are available when a developer manually enables the light bulb over a piece of code—for example, when right-clicking the code editor and selecting Quick Actions. Available refactorings depend on the kind of the selected syntax node. This is different from code analyzers, which report a warning or an error message by placing squiggles in the code editor and providing information in the Error List window.

You create a code refactoring using the **Code Refactoring (VSIX)** project template, which is available in the Extensibility node under the programming language of your choice in the New Project dialog (see Figure 34). This is actually the same location of the project template for code analyzers.

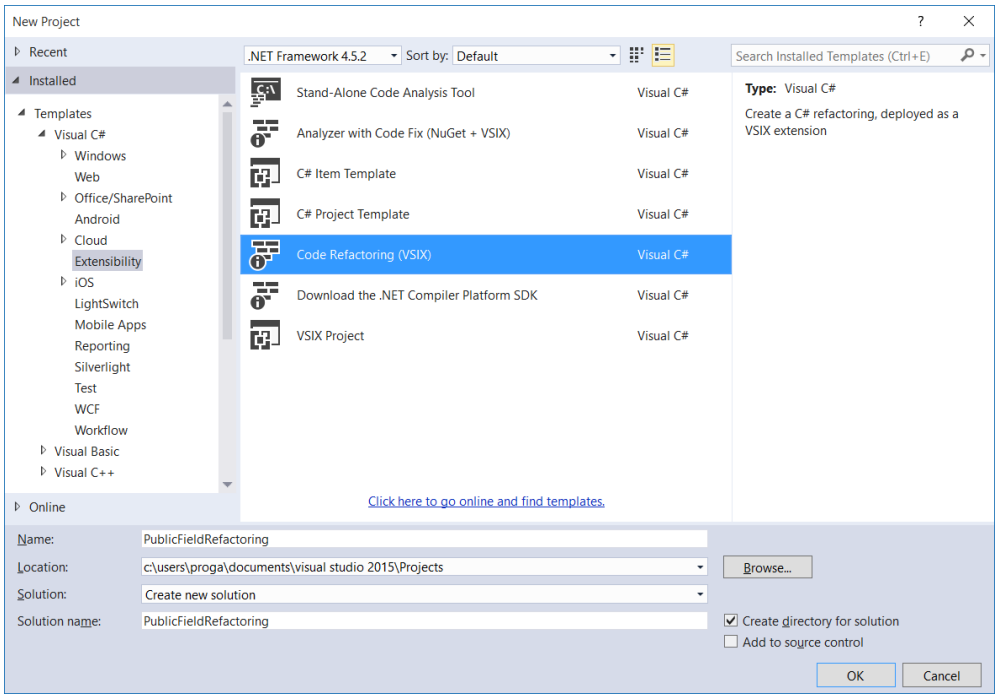

Figure 34: Creating a code refactoring project

There are many scenarios where code refactorings fit well. For instance, you can rewrite a piece of code to improve maintainability and readability, or you could automate the implementation of an interface within a class definition.

In this chapter, you'll learn how to create a custom refactoring that targets public fields and checks whether the first letter in an identifier's declaration is uppercase. If it isn't, the refactoring provides a quick way to rewrite the field declaration where the first character of its identifiers are uppercase. This is useful for addressing a requirement in the .NET Framework Design Guidelines, which state that identifiers of public members should be named using the Pascal casing notation, where the first character is uppercase.

To start, create a new project using the **Code Refactoring (VSIX)** template and name it **PublicFieldRefactoring** (see Figure 34).

Note: Unlike code analyzer projects, the Code Refactoring project template does not generate a NuGet package. Therefore, the only default way to publish a code refactoring is by using a VSIX package. In Chapter 6, "Deploying Analyzers to NuGet," you'll learn some tricks for including a code refactoring in a NuGet package.

Structure of Code Refactorings

When you create a code refactoring project, Visual Studio generates two projects:

- The actual code refactoring project, which is a portable class library. The compiled library will expose the custom code refactoring to development tools like Visual Studio. As with analyzers, this kind of project can contain multiple code refactorings.
- An extensibility project that generates a VSIX package for deploying the code refactoring library. The generated VSIX package is also deployed to the experimental instance of Visual Studio for debugging purposes. Chapter 7 describes how to share code refactorings with other developers by publishing the VSIX package to the Visual Studio Gallery.

The project contains a sample refactoring that works against type names and offers to reverse characters in the type name. Even though it's not very useful in real world scenarios, this can certainly be useful for instructional purposes, so it's worth taking a look at the code. By default, code refactorings are implemented in the CodeRefactoringProvider.cs (or .vb) code file. This is a conventional file name, and can be renamed with a more meaningful name; you will typically provide meaningful file names when you add multiple code refactorings to a single project.

Tip: A code refactoring project can expose an arbitrary number of custom code refactorings. If you want to add other code refactorings to the current project, right-click its name in Solution Explorer, select** Add **>** New Item**, and in the** Add New Item **dialog, select the** Refactoring **item template available in the** Extensibility **node.

A code refactoring is a class that inherits from **`Microsoft.CodeAnalysis.CodeRefactorings.CodeRefactoringProvider`**, which provides the common refactoring infrastructure. Such a class must be decorated with the **`ExportCodeRefactoringProvider`** attribute, which takes two arguments: the language the

code refactoring is designed for, and its name. The main entry point for a code refactoring is an asynchronous method called **`ComputeRefactoringsAsync`**, which is the place where you implement the refactoring logic. You can examine a code refactoring's structure by opening the auto-generated CodeRefactoringProvider.cs (or .vb) file and looking at the implementation of the sample refactoring, including comments. In this chapter, you will rewrite the code refactoring from scratch, ignoring the sample offered by Visual Studio.

Understanding Syntax Elements

Before you write a code refactoring that allows replacing the first letter of identifiers in a public field with an uppercase letter, you need to understand which syntax element you need to work with, and this is something you do again with the Syntax Visualizer tool. Visual Basic and C# have different syntax to represent a field declaration, and this is important because the sample code will be slightly different for the two languages.

Figure 35 shows the Syntax Visualizer over a C# field declaration, whereas Figure 36 shows it over a Visual Basic field declaration.

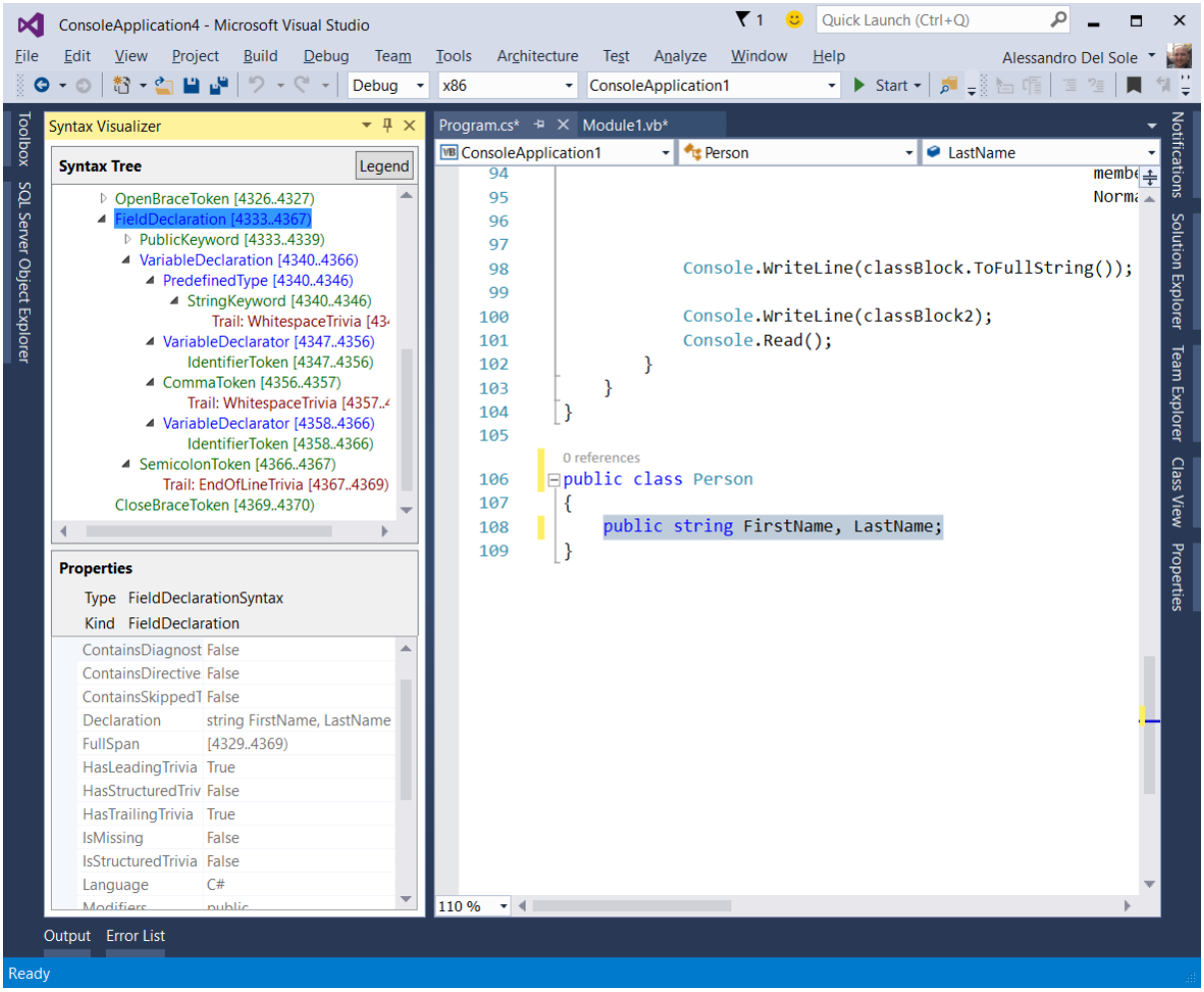

Figure 35: The syntax representation of a field declaration in C#

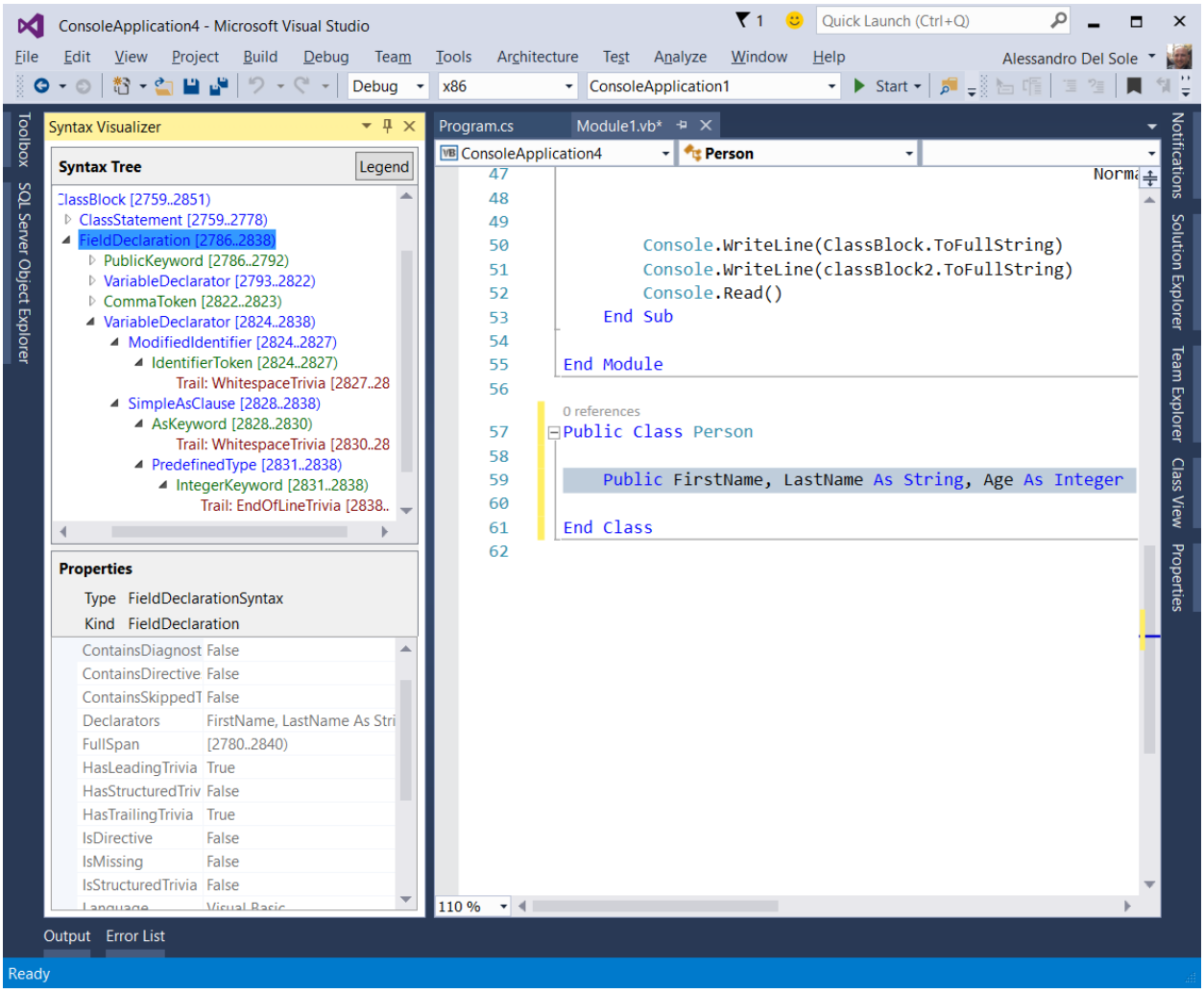

Figure 36: The syntax representation of a field declaration in Visual Basic

These are the most important considerations at this point:

- A field declaration is a FieldDeclaration syntax node, mapped by the **`FieldDeclarationSyntax`** class in both languages.
- In C#, **`FieldDeclarationSyntax`** exposes a property called **`Declaration`** of type **`VariableDeclarationSyntax`**, which exposes a property called **`Variables`**, a collection of **`VariableDeclaratorSyntax`** objects, each representing an identifier declaration in the field.
- In Visual Basic, **`FieldDeclarationSyntax`** exposes a property called **`Declarators`**, a collection of **`VariableDeclaratorSyntax`** objects, each representing an identifier declaration in the field.

Implementing the Refactoring Logic

In Visual Basic, you need to iterate the **`FieldDeclarationSyntax.Declarators`** property and search for any identifier that starts with a lowercase letter; in C#, you need to iterate the

FieldDeclarationSyntax.Declaration.Variables property with the same goal. Common to both languages, you need a method that converts the identifier name the appropriate way, and you need to create and register an action, similarly to what you did with code fixes in the previous chapter.

To create an action, you invoke the **Microsoft.CodeAnalysis.CodeActions.CodeAction.Create** static method (the same used with code fixes) and register the action, invoking the **RegisterRefactoring** method exposed by the refactoring context instance.

Code Listing 16 shows how to implement the **ComputeRefactoringAsync** method. Notice that from now on, since the code re-uses many concepts described in Chapter 4, I will not explain these concepts in the text, but detailed comments have been added to the code to aid your understanding.

Code Listing 16 (C#)

```
public sealed override async Task
    ComputeRefactoringsAsync(
    CodeRefactoringContext context)
{
    // Get the root syntax node
    var root = await context.Document.
        GetSyntaxRootAsync(context.CancellationToken).
        ConfigureAwait(false);

    // Find the node at the selection
    var node = root.FindNode(context.Span);

    // Convert the node into a field declaration
    var fieldDecl = node as FieldDeclarationSyntax;
    // If it's not public, return
    if (fieldDecl == null ||
        fieldDecl.Modifiers.ToFullString().
        Contains("public") == false) { return; }

    // Used to determine if an action must
    // be registered
    bool mustRegisterAction = false;

    // If at least one starting character is
    // lowercase, must register an action
    if (fieldDecl.Declaration.
        Variables.Any(i =>
        char.IsLower(i.Identifier.
        ValueText.ToCharArray().First())))
    {
        mustRegisterAction = true;
    }
    else
```

```
            {
                mustRegisterAction = false;
            }


            if (mustRegisterAction==false)
            {
                return;
            }

            // Create an action invoking a delegate passing
            // the document instance, the syntax node, and
            // a cancellation token
            var action =
                CodeAction.Create("Make first char upper case",
                           c => RenameFieldAsync(context.
                           Document, fieldDecl, c));

            // Register this code action
            context.RegisterRefactoring(action);
        }
```

Code Listing 16 (VB)

```
Public NotOverridable Overrides Async _
        Function ComputeRefactoringsAsync(
                 context As CodeRefactoringContext) As Task

        'Get the root syntax node
        Dim root = Await context.Document.
            GetSyntaxRootAsync(context.
            CancellationToken).ConfigureAwait(False)

        'Find the node at the selection
        Dim node = root.FindNode(context.Span)

        'Convert the node to a field declaration
        Dim fieldDecl = TryCast(node,
            FieldDeclarationSyntax)

        'If it's not public, return
        If fieldDecl Is Nothing Or fieldDecl.Modifiers.
            ToFullString.Contains("Public") = False Then
            Return
        End If

        'Used to determine if an action
        'must be registered
```

```
        Dim mustRegisterAction As Boolean

        'Iterate the variable declarators
        For Each declarator
            In fieldDecl.Declarators

            'If at least one starting character
            'is lowercase, must register an action
            If declarator.Names.Any(Function(d) _
                        Char.IsLower(d.Identifier.
                        Value.ToString(0))) Then
                mustRegisterAction = True
            Else
                mustRegisterAction = False
            End If
        Next

        If mustRegisterAction = False Then
            Return
        Else
            'Create an action invoking a delegate passing
            'the document instance, the syntax node, and
            'a cancellation token
            Dim action =
                CodeAction.Create("Make first char upper case",
                           Function(c) RenameFieldAsync(context.
                           Document, fieldDecl, c))

            ' Register this code action
            context.RegisterRefactoring(action)
        End If

    End Function
```

The next step is to generate a new syntax node for the selected field declaration, supplying updated variable identifiers with the first letter, now uppercase. This is demonstrated in Code Listing 17.

Code Listing 17 (C#)

```
        private async Task<Document>
            RenameFieldAsync(Document document,
            FieldDeclarationSyntax fieldDeclaration,
            CancellationToken cancellationToken)
        {
            // Get the semantic model for the code file
            var semanticModel =
```

```
            await document.
                  GetSemanticModelAsync(cancellationToken).
                  ConfigureAwait(false);

        // Get the root syntax node
        var root = await document.GetSyntaxRootAsync();

        // Get a list of old variable declarations
        var oldDeclarators =
            fieldDeclaration.Declaration.
            Variables;

        // Store the collection of variables
        var listOfNewModifiedDeclarators =
            new SeparatedSyntaxList<
                VariableDeclaratorSyntax>();

        // Iterate the declarators collection
        foreach (var declarator in oldDeclarators) {
            // Get a new name
            var tempString =
                ConvertName(declarator.Identifier.
                ToFullString());

            // Generate a new modified declarator
            // based on the previous one but with
            // a new identifier
            listOfNewModifiedDeclarators =
                listOfNewModifiedDeclarators.
                Add(declarator.WithIdentifier(
                    SyntaxFactory.ParseToken(tempString)));
        }

        // Generate a new field declaration
        // with updated variable names
        var newDeclaration =
            fieldDeclaration.Declaration.
            WithVariables(listOfNewModifiedDeclarators);

        // Generate a new FieldDeclarationSyntax
        var newField = fieldDeclaration.
            WithDeclaration(newDeclaration);

        // Replace the old syntax node with the new one
        SyntaxNode newRoot = root.
                    ReplaceNode(fieldDeclaration,
                    newField);

        // Generate a new document
```

```
            var newDocument =
                document.WithSyntaxRoot(newRoot);

            // Return the document
            return newDocument;
        }

        // Return a new identifier with an
        // uppercase letter
        private string ConvertName(string oldName)
        {
            return char.
                   ToUpperInvariant(oldName[0]) +
                   oldName.Substring(1);
        }
```

Code Listing 17 (VB)

```
    Private Async Function _
        RenameFieldAsync(document As Document,
                         fieldDeclaration As FieldDeclarationSyntax,
                         cancellationToken As CancellationToken) _
                         As Task(Of Document)

        'Get the semantic model for the code file
        Dim semanticModel = Await document.
            GetSemanticModelAsync(cancellationToken).
            ConfigureAwait(False)

        'Get the root syntax node
        Dim root = Await document.GetSyntaxRootAsync

        'Get a list of old declarators
        Dim oldDeclarators =
            fieldDeclaration.Declarators

        'Store the collection of identifiers
        Dim listOfNewModifiedIdentifiers As _
            New SeparatedSyntaxList(
            Of ModifiedIdentifierSyntax)
        'Store the collection of declarators
        Dim listOfNewModifiedDeclarators As _
            New SeparatedSyntaxList(
            Of VariableDeclaratorSyntax)

        'Iterate the declarators collection
        For Each declarator In oldDeclarators
            'For each variable name in the declarator...
```

```
            For Each modifiedIdentifier In declarator.Names
                'Get a new name
                Dim tempString =
                    ConvertName(modifiedIdentifier.
                    ToFullString())

                'Generate a new ModifiedIdentifierSyntax based on
                'the previous one's properties but with a new Identifier
                Dim newModifiedIdentifier As _
                    ModifiedIdentifierSyntax =
                    modifiedIdentifier.
                    WithIdentifier(SyntaxFactory.
                    ParseToken(tempString)).
                    WithTrailingTrivia(modifiedIdentifier.
                    GetTrailingTrivia)

                'Add the new element to the collection
                listOfNewModifiedIdentifiers =
                    listOfNewModifiedIdentifiers.
                    Add(newModifiedIdentifier)
            Next
            'Store a new variable declarator with new
            'variable names
            listOfNewModifiedDeclarators =
                listOfNewModifiedDeclarators.Add(declarator.
                WithNames(listOfNewModifiedIdentifiers))

            'Clear the list before next iteration
            listOfNewModifiedIdentifiers = Nothing
            listOfNewModifiedIdentifiers = New _
                SeparatedSyntaxList(Of ModifiedIdentifierSyntax)
        Next

        'Generate a new FieldDeclarationSyntax
        Dim newField = fieldDeclaration.
            WithDeclarators(listOfNewModifiedDeclarators)

        'Replace the old declaration with the new one
        Dim newRoot As SyntaxNode =
            root.ReplaceNode(fieldDeclaration,
                             newField)

        'Generate a new document
        Dim newDocument =
            document.WithSyntaxRoot(newRoot)

        'Return the new document
        Return newDocument
    End Function
```

```
    'Return a new identifier with an
    'uppercase letter
    Private Function _
        ConvertName(oldName As String) As String
        Return Char.
            ToUpperInvariant(oldName(0)) &
            oldName.Substring(1)

    End Function
```

As you can see, you have re-used skills you acquired in Chapter 4 to create new syntax elements for updating a FieldDeclaration syntax node. One new thing you saw was how to create instances of collections that are specific to Roslyn (such as **`SeparatedSyntaxList`**) without using the **`SyntaxFactory`** class.

Testing and Debugging a Code Refactoring

You test and debug a code refactoring exactly as you do with code analyzers, so you just need to make sure the .vsix project is selected as the startup project, and then press F5. The code refactoring library will be deployed to the experimental instance of Visual Studio, and will be ready for your tests. Of course, you can definitely use the debugging instrumentation you already know. Figure 37 demonstrates how the custom refactoring works over a field declaration in Visual Basic (you get the same result in C#).

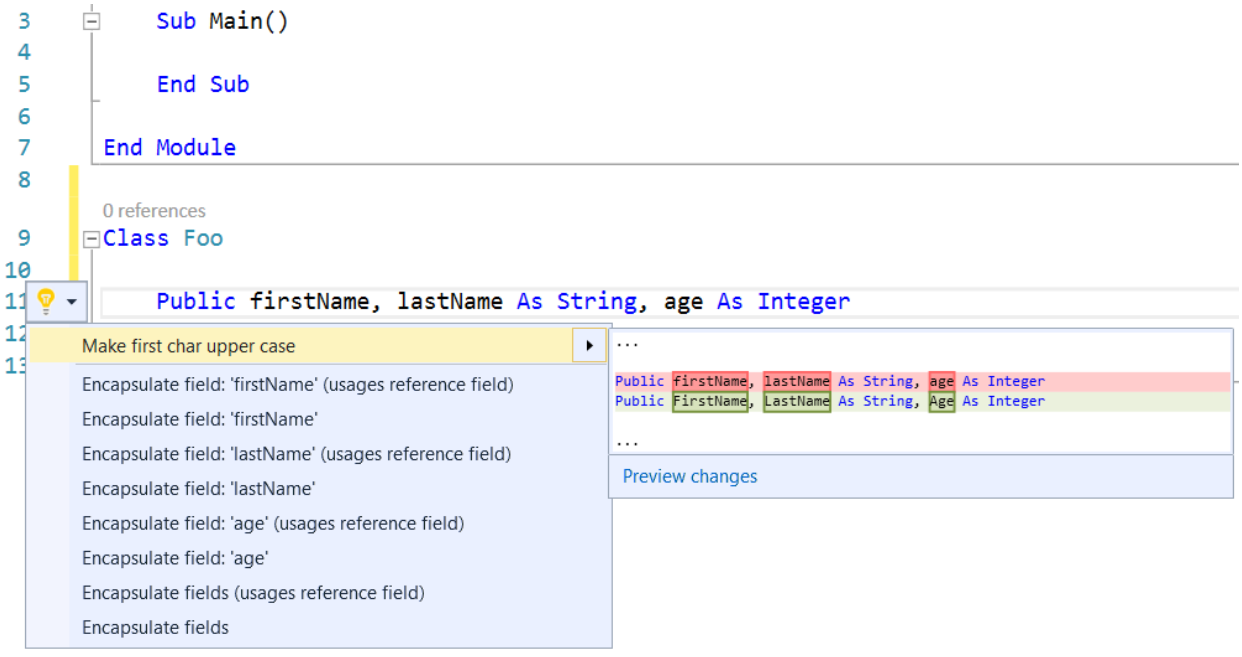

Figure 37: Testing a code refactoring

Custom code refactorings definitely improve the coding experience, not only because they allow reorganizing portions of code a better way, but also because they can simplify the implementation of programming patterns. As an example, imagine you want to offer a refactoring that automates the generation of the **`RelayCommand`** and **`ViewModelBase`** classes in an implementation of the Model-View-ViewModel pattern. You could create a code refactoring that rewrites the syntax tree of a class by adding the syntax nodes required for the pattern implementation.

Tip: You can take a look at an open-source project I created and published on GitHub, called DotNetAnalyzers. Among others, this project has code refactorings in both VB and C# that automate the generation of classes that implement the MVVM pattern.

Chapter Summary

Refactoring code means reorganizing portions of code a better way without changing the original behavior. In this chapter, you saw how to create a custom code refactoring that integrates with Visual Studio's quick actions, you saw how to use the Syntax Visualizer to determine the syntax nodes you need to edit, you learned about how a code refactoring is architected, and you tried out the basics for implementing the refactoring logic. Finally, you saw how to test and debug a code refactoring locally. So far, you have learned many concepts about the .NET Compiler Platform, and now you have the skills you need to create code analyzers and refactorings on your own. However, you have only used the result of your work on your own, locally. The big benefit of Roslyn is that code analyzers and refactorings can be shared with other developers. This is discussed in the next two chapters.

Chapter 6 Deploying Analyzers to NuGet

One of the benefits of code analyzers is that they can be shared with other developers. For instance, you might have written an analyzer that detects code issues when implementing the MVVM pattern in WPF, and you want to make it public. Additionally, you might ship your own libraries and APIs including integrated Roslyn code analysis. This is an incredible opportunity: For example, if you build and sell user controls, you can include code analyzers that detect code issues and suggest fixes based on the programming patterns required by your APIs. This chapter explains how to package and deploy analyzers to the popular online library repository NuGet for the best integrated experience possible.

Note: The steps described in this chapter do not apply to code refactoring projects because these do not automate the generation of a NuGet package. Some tricks to publish a code refactoring to NuGet will be provided in this chapter, but the official, supported way to deploy a refactoring is only via the Visual Studio Gallery, which is the topic of the next chapter.

Quick Recap: About NuGet

NuGet (pronounced *New Get*) is an online package repository where developers can publish and find libraries. A NuGet package contains .dll libraries and metadata information about the package description and package dependencies. Visual Studio 2015 has integrated NuGet client tools that allow finding, downloading, and installing NuGet packages from within the IDE. These client tools automatically resolve and install any dependencies the package relies on, so that you do not have to worry about finding and downloading other libraries manually. You saw an example of how to use NuGet to install an analyzer package to a project in Chapter 2. In this chapter, you make a step forward: you will learn how to publish an analyzer package to NuGet, so that other developers will be able to install and use your analysis rules in their projects.

Preparing an Example

To demonstrate the process of deploying an analyzer to NuGet, I will use the DateTimeAnalyzer project created in Chapter 4. Obviously, you will not be able to publish the same analyzer because it already exists on the NuGet gallery, but the steps described here apply to any analyzers you build on your own.

Preparing for Publication

The **Analyzer with Code Fix (NuGet + VSIX)** project template automatically generates a NuGet package containing the compiled analyzer every time you build the project. This is possible because of specific MSBuild rules that automate the generation of the package. This resides in the Bin\Debug or Bin\Release subfolders of your project, according to the selected build

configuration. Its name is made of the project name, of the version number, and of the .nupkg extension. For example, the name for latest stable version of the DateTimeAnalyzer package I'm using for this chapter is called DateTimeAnalyzer_VB.1.0.5710.29175.nupkg, where 1.0 is the major/minor pair and 5710.29175 is the build/revision pair.

Understanding NuGet Packages for Analyzers

Generally speaking, NuGet packages are zip archives with .nupkg extensions, and contain a number of elements, including files that must be installed onto the target machine. The structure for a NuGet package can be very complex, and explaining it in detail is out of the scope of this book, whose purpose is teaching you the basics of the .NET Compiler Platform, not the NuGet package conventions.

With specific regard to analyzers, a NuGet package contains (at least) the following elements:

- The package manifest, which contains metadata information and the list of files and dependencies that the package relies on. This is discussed in the next subsection.
- A folder called **analyzers\dotnet** that contains language-agnostic analyzers or language-specific folders, such as **analyzers\dotnet\vb** and **analyzers\dotnet\cs**.
- A folder called **tools**, which contains two PowerShell scripts called **Install1.ps** and **Uninstall1.ps**. When you install an analyzer package to a project with the NuGet Package Manager, Visual Studio 2015 executes the **Install1.ps** script to install the analyzer. To remove the package, Visual Studio executes the **Uninstall.ps1** script when you decide you do not need it any longer and you want to remove it via the NuGet tooling.

Understanding these basic concepts is important because you will work with the structure of a package when editing its manifest in the next subsection. For a comprehensive list of elements that can be added to a NuGet package, and for a full package authoring guide, you can read the official NuGet documentation.

Editing the Package Manifest

Before a NuGet package can be published, you need to edit its manifest. The manifest contains metadata information, such as package id, title, description, and author, but also the list of files that the package is going to install to the target project and the list of dependencies that NuGet will automatically resolve. The package manifest is an XML file with .nuspec extension. For analyzers, Visual Studio 2015 automatically creates a manifest file called **Diagnostic.nuspec** that you can see in Solution Explorer and open inside a Visual Studio editor window. The XML elements in the .nuspec file are self-explanatory; however, you must pay particular attention to the following:

- **`id`**: this represents the unique identifier for the package in the NuGet online gallery.
- **`version`**: this allows specifying the package version number and is important for providing package updates. Notice that you can manually edit the major and minor version numbers, but Visual Studio 2015 automatically supplies and updates the build and revision version numbers of a package every time you rebuild the project, disregarding the content of **`version`**.
- **`licenseUrl`**: the web address of the license agreement for the package (optional).

- **requireLicenseAcceptance**: if **true**, users will need to accept the linked license agreement before downloading and installing the package. Notice that the default value is **false**. Remember that it cannot be **true** if you do not specify a valid license URL.
- **tags**: by specifying comma-separated tags, users will be able to find your analyzer in the NuGet gallery by typing one of the specified tags in either the search box of the Visual Studio's Manage NuGet Packages window, or in the online web site.

Code Listing 18 shows how the Diagnostic.nuspec file has been edited for the current sample package.

Code Listing 18: Editing the NuGet package manifest

```
<?xml version="1.0"?>
<package
xmlns="http://schemas.microsoft.com/packaging/2011/08/nuspec.xsd">
  <metadata>
    <id>DateTimeAnalyzer_VB</id>
    <version>1.0.0.0</version>
    <title>DateTime analyzer for Visual Basic</title>
    <authors>Alessandro Del Sole</authors>
    <owners>Alessandro Del Sole</owners>
    <licenseUrl>http://opensource.org/licenses/MIT</licenseUrl>
    <projectUrl>https://github.com/AlessandroDelSole/RoslynSuccinctly/>
    <!-- Commenting this line, not necessary
    <iconUrl>http://ICON_URL_HERE_OR_DELETE_THIS_LINE</iconUrl> -->
    <requireLicenseAcceptance>true</requireLicenseAcceptance>
    <description>This analyzers detects improper usages of the DateTime
type in Windows Store and OData applications, where you should use
DateTimeOffset instead</description>
    <releaseNotes>First stable release to the public.</releaseNotes>
    <copyright>Copyright 2015, Alessandro Del Sole</copyright>
    <tags>Roslyn, analyzers, datetime</tags>
  </metadata>
  <!-- The convention for analyzers is to put language agnostic dlls in
  analyzers\dotnet and language specific analyzers in either
  analyzers\dotnet\cs or analyzers\dotnet\vb -->
  <files>
    <file src="*.dll" target="analyzers\dotnet\vb"
          exclude="**\Microsoft.CodeAnalysis.*;
**\System.Collections.Immutable.*;
**\System.Reflection.Metadata.*;
**\System.Composition.*" />
    <file src="tools\*.ps1" target="tools\" />
  </files>
</package>
```

You can use the **releaseNotes** element to describe what's new with an updated package version. Notice how the **files** node contains the list of files that will be installed by the package. More specifically, the first file element will install the analyzer's .dll library into a project folder called analyzers\dotnet\vb if the analyzer targets Visual Basic (VB), or into

analyzers\dotnet\cs if the analyzer targets C#. If the analyzer is language-agnostic (that is, can be used against both languages), the analyzer's .dll library is installed into a project folder called analyzer\dotnet. The **`exclude`** attribute tells NuGet that the specified dependencies will not be downloaded (the reason is that they are available by default in a project). The second **`file`** element tells NuGet to extract the Install.ps1 and Uninstall.ps1 scripts described in the previous section into a project subfolder called **tools**.

Assuming you have made all the required edits to the manifest of your analyzer, make sure the build configuration is set to **Release**, and then rebuild the project. Open the **Bin\Release** project subfolder and locate the highest updated version of your package. This will be used in the next section to complete the deployment process.

Publishing Analyzers to NuGet

In the previous section, you created a NuGet package for your analyzer. Such a package can be published to the NuGet gallery so that other developers will be able to consume it in their projects in the same way you did with the DateTimeAnalyzer in Chapter 2. However, the official NuGet repository is not a playground, and you might want to avoid publishing unstable packages. Fortunately, Visual Studio 2015 can consume local packages on your development machines. This allows local testing before graduating to the online repository. This is the next topic of this chapter.

Testing Packages Locally

Visual Studio 2015 allows picking NuGet packages not only from the official NuGet repository online, but also from local folders and online feeds that are compatible with the NuGet specifications. This gives you an option to test your packages on your development machine before publishing your work online. To demonstrate how this works, create a new local folder called **C:\LocalPackages**. Then, copy the DateTimeAnalyzer_CS.1.0.5710.29175.nupkg analyzer's NuGet package into this folder (or DateTimeAnalyzer_VB.1.0.5710.29175.nupkg for Visual Basic).

Finally, open Visual Studio 2015. Remember that Visual Studio updates the package version number at every build, so you might have a slightly different version number. What you have to do at this point is configure the integrated NuGet client tools to recognize the local folder as a package source. To accomplish this, select **Tools** > **Options**, and in the Options dialog box, expand the **NuGet Package Manager** node. Finally, select the **Package Sources** element (see Figure 38).

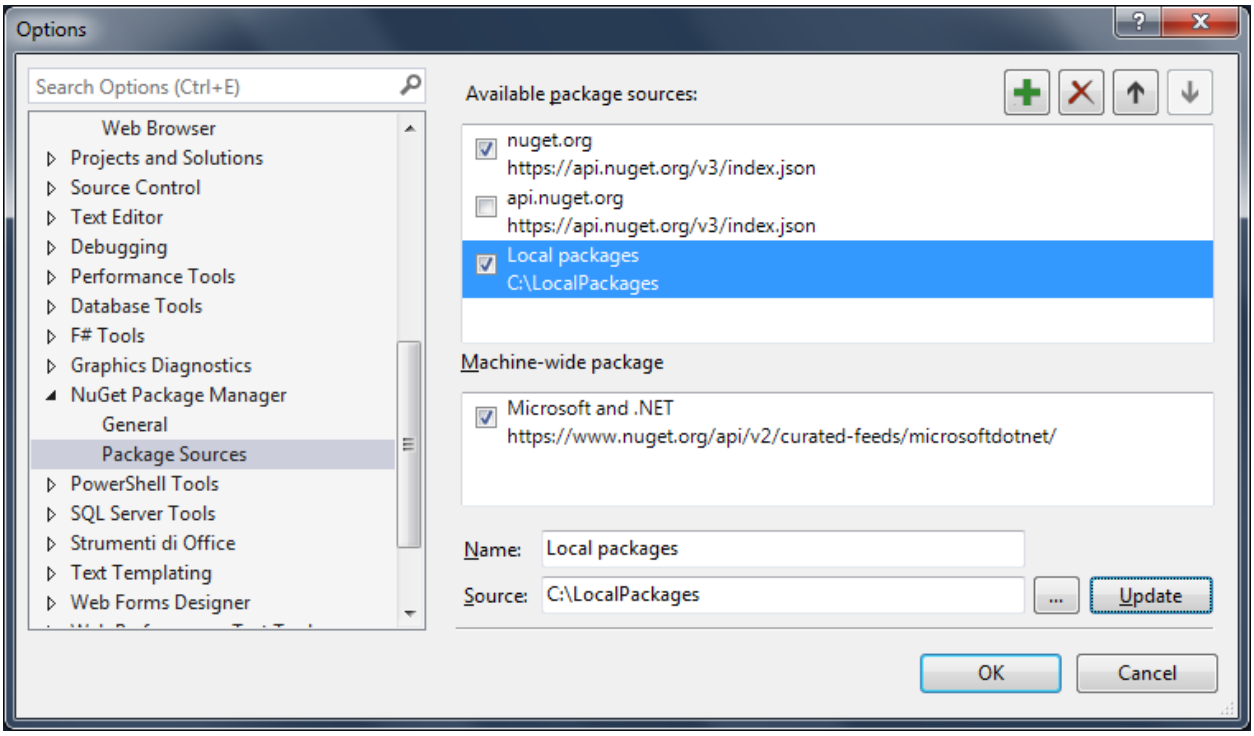

Figure 38: Configuring NuGet package sources

Click the button with the green **+** symbol to add a new package source. In the Name text box, enter a friendly name for the package source, which will be displayed in the NuGet Package Manager window. In the Source text box, enter the newly created folder name (see Figure 38). When ready, click **OK**.

To test the package locally, follow these steps:

1. Create and save a new Windows Phone 8.1 project with the programming language that matches the analyzer's target language.
2. In Solution Explorer, right-click the project name and select **Manage NuGet Packages**.
3. When the NuGet Package Manager appears, from the **Package source** combo box, select the **Local packages** feed (see Figure 39).
4. Select the desired package, review the information, and then click **Install**.

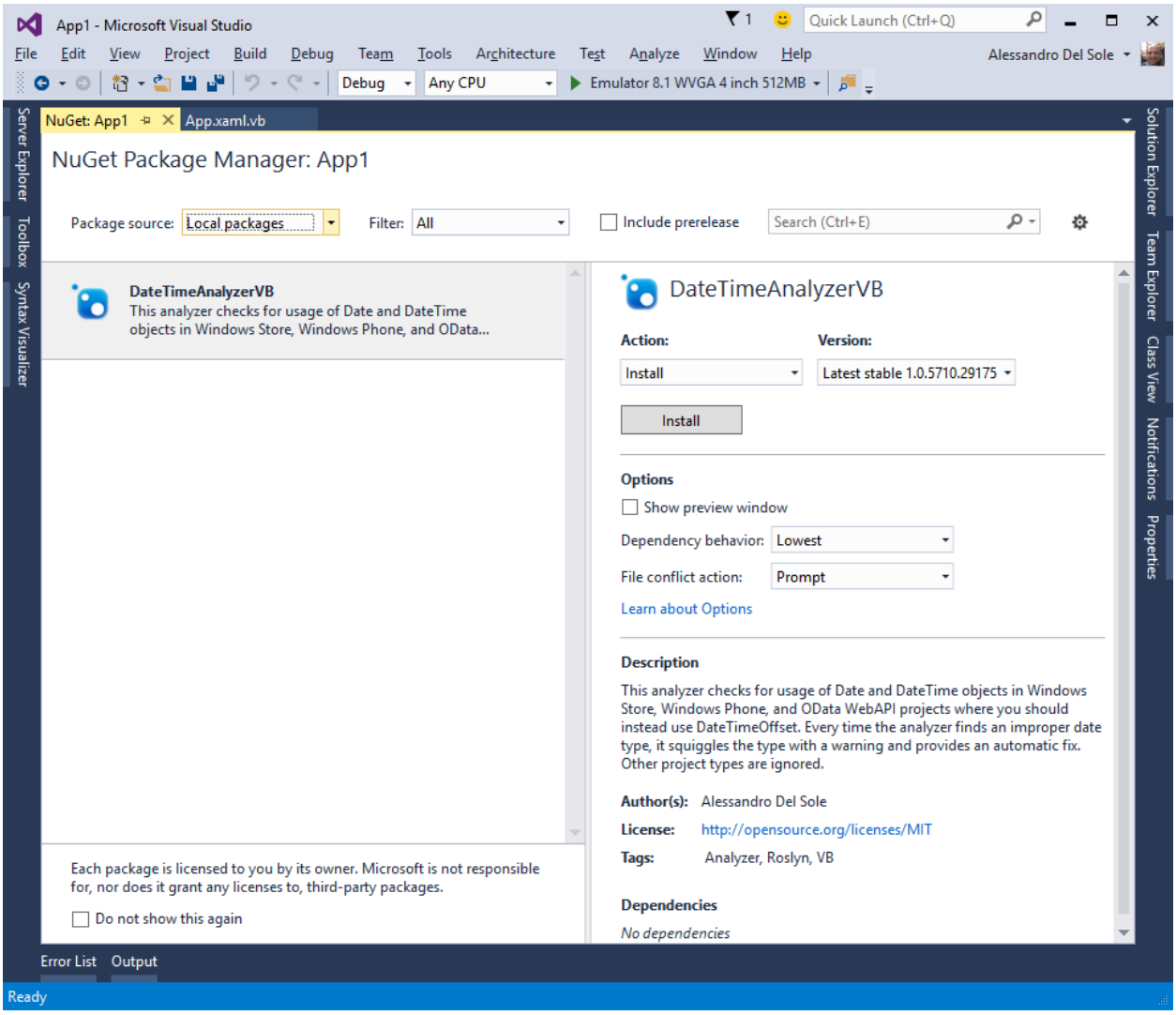

Figure 39: Selecting and installing the NuGet package from a local repository

After a few seconds, the package will be installed to the project and the analyzer will be immediately available. To demonstrate how it works, you can follow the steps described in the section called “Downloading, Installing, and Using Code Analyzers” in Chapter 2, and you can take Figures 17 and 18 as a reference. Assuming you have made your local tests to make sure the analyzer works as expected, it’s time to move online and share the analyzer with other developers.

Note: Developers expect to find only stable and professional packages on the official NuGet gallery. So before you graduate your package from local testing to NuGet, it’s a good idea to make an intermediate test by publishing the package to one of the online NuGet hosting services, such as MyGet. This allows you to create your personal and enterprise NuGet feeds outside of the official gallery.

Moving to the Online NuGet Repository

In order to share an analyzer package with the public, you have to sign into the NuGet portal. If you don't have an account, you can register for free by supplying your credentials or using an existing Microsoft account.

Once you've signed in, click **Upload Package**. You will be asked to specify the NuGet package you want to upload, so click **Browse**, and then locate and select the highest version of the package. Figure 40 shows an example based on version 1.0.5710.29175 of the DateTimeAnalyzer_VB package.

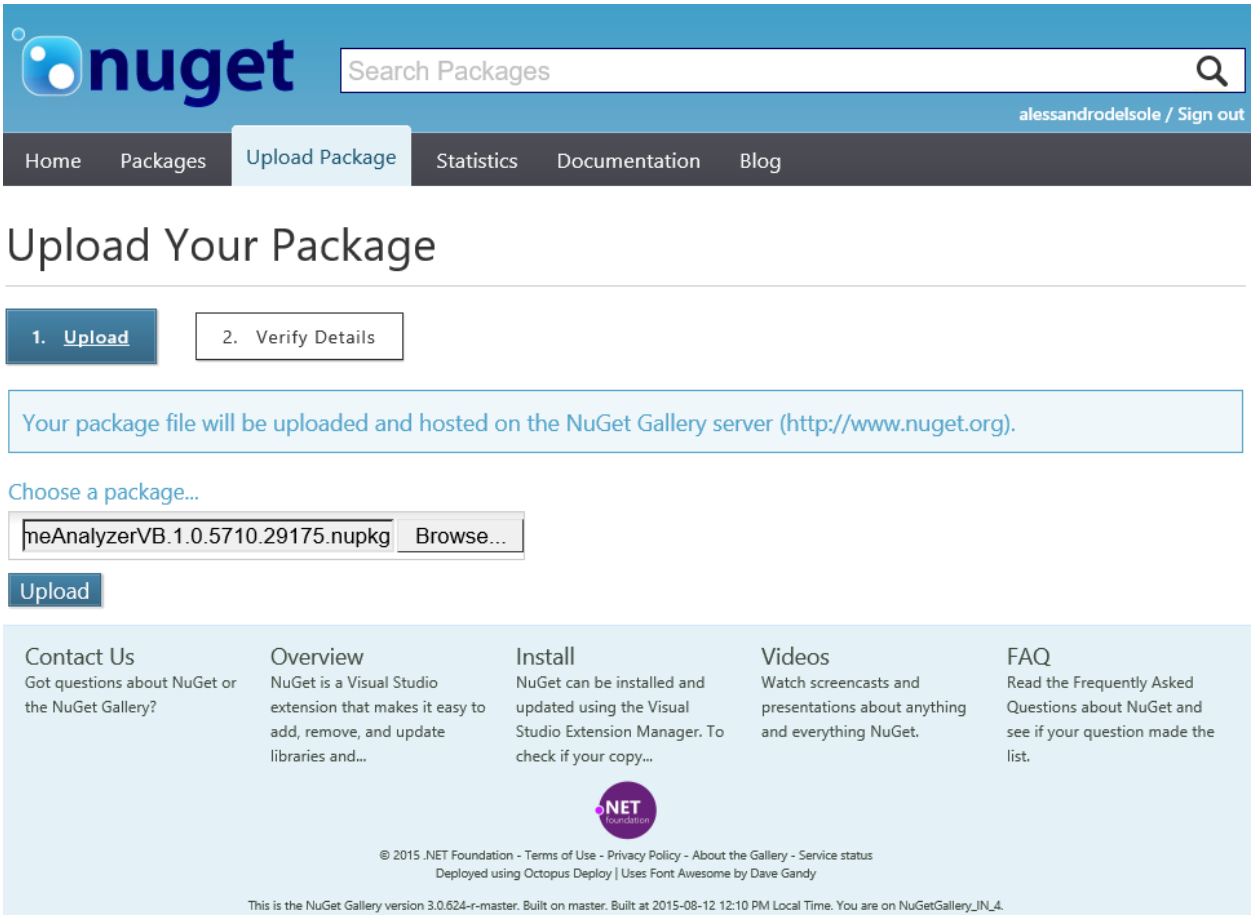

Figure 40: Uploading a NuGet package

Click **Upload**. When the upload has been completed, you will have an option to review the package details, as shown in Figure 41. Whether or not you need to make changes to the package details, submit the information to finalize the publication process. After 15 to 20 minutes, the package will be listed in the NuGet feed.

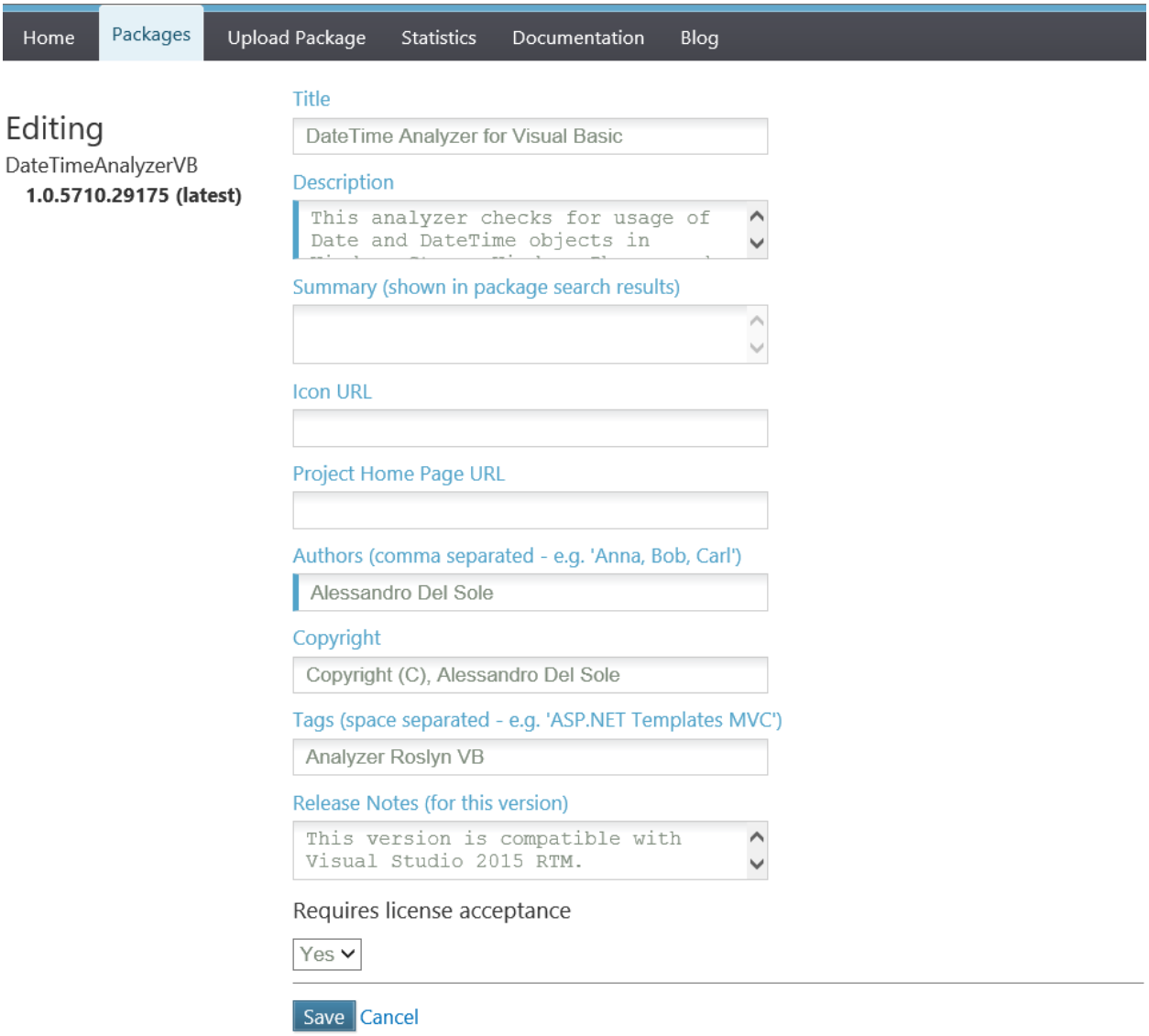

Figure 41: Viewing and editing package details

In Visual Studio 2015, you can then add the NuGet package from the official repository in one of the supported project types, for example, a Windows Phone 8.1 app. Figure 42 shows the NuGet Package Manager window with the published package highlighted.

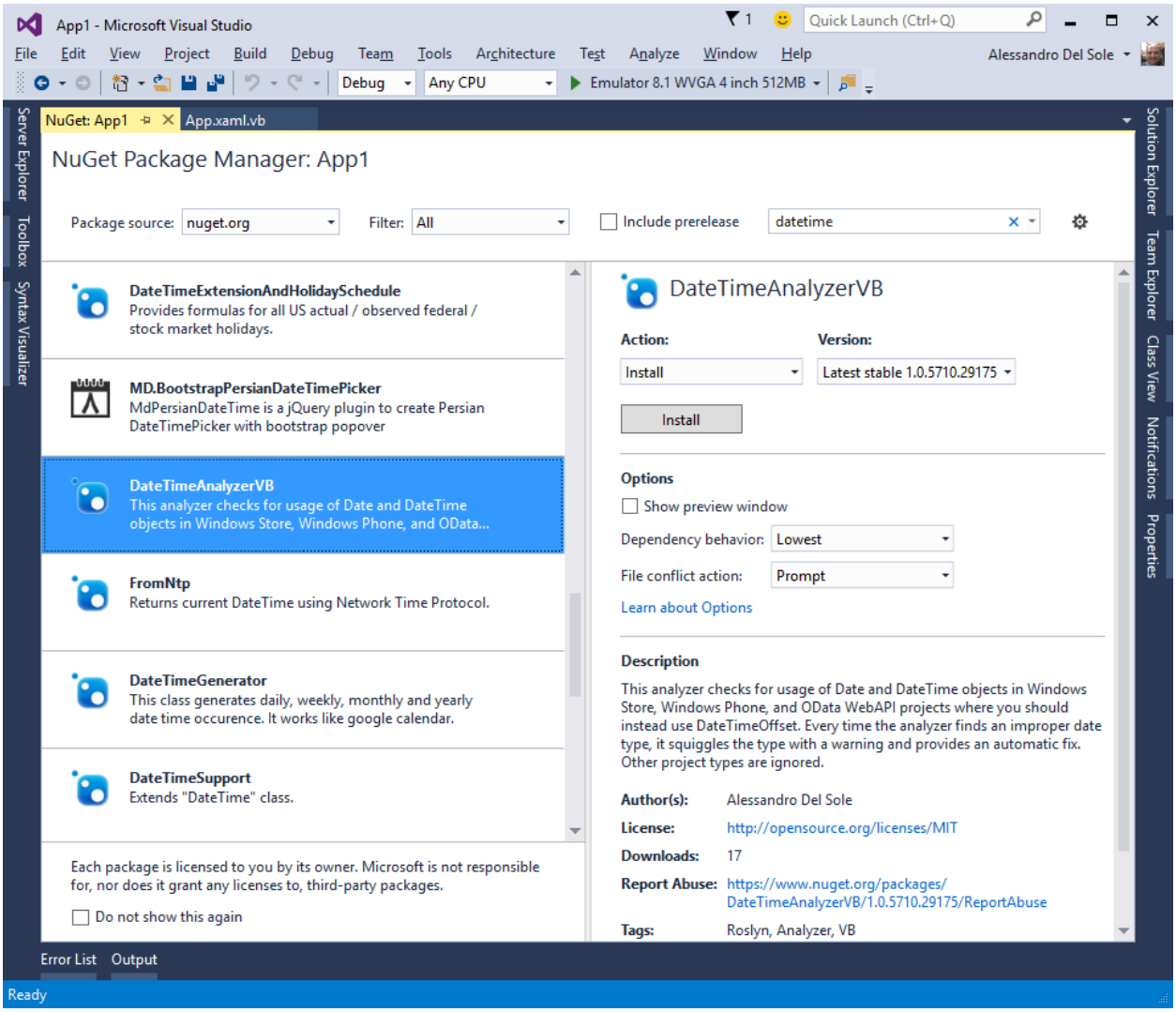

Figure 42: The NuGet package is available from the online repository

You can then follow the steps described in Chapter 2 to download the package and use the analyzer it will install to your project. At this point, you have shared your analyzer with the world, and other developers will be able to leverage your analysis rules in their projects.

Tips and Tricks for Analyzers and Refactorings

This section includes a number of tips and tricks to make your experience with NuGet and Roslyn better.

Deploying Refactorings via NuGet Packages

As I told you at the beginning of this chapter, the Code Refactoring project template does not include MSBuild rules that automate the generation of a NuGet package, and the default way to

share a refactoring is via the Visual Studio Gallery, which is detailed described in the next chapter. However, if you add a code refactoring to an analyzer project via the Refactoring item template, this is bundled into the NuGet package together with the analyzer, and will be properly installed alongside the diagnostics.

Integrating Roslyn Analysis with Your APIs

You can ship NuGet packages that contain your libraries and Roslyn analyzers and refactorings. This gives developers an improved coding experience when using your APIs. For instance, a diagnostic might report a warning every time a developer is writing improper code against your libraries, and a code fix might help supplying more appropriate code snippets. You have two alternatives to accomplish this:

- Including your .dll libraries into the NuGet package that Visual Studio generates for your analyzers.
- Publish two separate NuGet packages: one for your APIs, and one for code analysis, plus a third, empty package that only contains metadata and dependencies information required to gather them both at install time. This approach allows developers to choose if they only want your APIs or both libraries and integrated code analysis.

In both cases, you need to be familiar with package authoring. The best place to find information about this is the official NuGet documentation. Also, you can take a look at an article I wrote for MSDN Magazine, which discusses this topic in more detail.

Chapter Summary

Microsoft Visual Studio 2015 provides an easy way to share your code analyzers with other developers by automating the generation of a NuGet package that contains the analyzer binaries and metadata information required to provide package details to consumers. Before you deploy your analyzers to the official NuGet gallery, it is important to make local tests on your development machine to make sure your package works properly and offers a stable development experience.

Visual Studio 2015 has specific options that can be configured to pick packages from local folders, so that you can test any packages as you would from the online NuGet repository. Once you are satisfied with local tests, you can graduate your package to NuGet by signing in and uploading your package. Developers will be able to find your analyzers in a few minutes, and they will be able to take advantage of your Roslyn-powered work.

Chapter 7 Deploying Analyzers and Refactorings to the Visual Studio Gallery

Note: The content of this chapter does not apply to Visual Studio 2015 Express editions.

For years, developers have used the Visual Studio Gallery for downloading third-party extensions for Visual Studio (via the integrated Extensions Manager tool) and publishing custom extensions so they can be consumed by other developers. Now, you can also publish code analyzers and refactorings to the Visual Studio Gallery in the form of Visual Studio extensions and make them available to other developers via the familiar IDE instrumentation.

Understanding VSIX Packages

When you create analyzers and refactorings, both the **Analyzer with Code Fix (NuGet + VSIX)** and the **Code Refactoring (VSIX)** templates include a project that automates the generation of a VSIX package, which you can easily see in Solution Explorer because its name ends with **.vsix**. VSIX packages are compiled, self-installing .vsix files that contain extensions for Visual Studio. If you ever installed an extension for Visual Studio from either the Visual Studio Gallery or the integrated Extensions Manager tool, you might already know that such an extension is deployed with a VSIX package. An extension for Visual Studio is a .dll library tailored to be plugged into the IDE that extends the environment with custom features such as new commands, custom tool windows, and more.

Behind the scenes, VSIX packages are Open XML compressed files that contain:

- The .dll library.
- A package manifest called extension.vsixmanifest that contains information such as the author information, version number, package description, package dependencies, and the list of Visual Studio editions the package is intended for.
- The license agreement that the user must accept in order to install the extension.
- An optional icon and preview image that will be used to present the extension in the Visual Studio Gallery and in the Visual Studio Extension and Updates dialog.

You can easily investigate the content structure of a VSIX package by opening a .vsix file with an archiver tool such as WinZip or WinRAR. Because both code analyzers and refactorings are .dll libraries that can be plugged into the code editor, they can be packaged into VSIX files and recognized as if they were Visual Studio extensions. Fortunately, Visual Studio 2015 automatically generates VSIX packages for you, making the deployment experience easier.

Note: When you press F5 to debug an analyzer or refactoring in the experimental instance of Visual Studio 2015, the IDE first installs the auto-generated VSIX package to the experimental instance. This is necessary to allow you to debug an analyzer or refactoring through the usual debugging instrumentation.

Preparing an Example

In this chapter, you will see how to deploy a code refactoring to the Visual Studio Gallery. This section looks at code refactorings due to the fact that the Visual Studio 2015 project template for code refactorings does not include support for NuGet, but only automates the generation of a VSIX package. On the other hand, the project template for code analyzers automates the generation of both a NuGet package and a VSIX package. As a result, the steps described in this chapter apply to both refactorings and analyzers. With that in mind, in Visual Studio 2015, open the **PublicFieldRefactoring** project you created back in Chapter 5. This will serve as the base for deployment.

Preparing for Publication

Before you publish a VSIX package, you need to edit some information included in the package manifest, which contains information such as the package description, author name, license agreement, release notes, icon and preview images, and links to documentation. The package manifest is called **source.extension.vsixmanifest**, and is available in the .vsix project. Editing the manifest is very easy since Visual Studio 2015 offers a convenient designer, which you enable by double-clicking **source.extension.vsixmanifest** in the Solution Explorer. Figure 43 shows how to edit the package manifest.

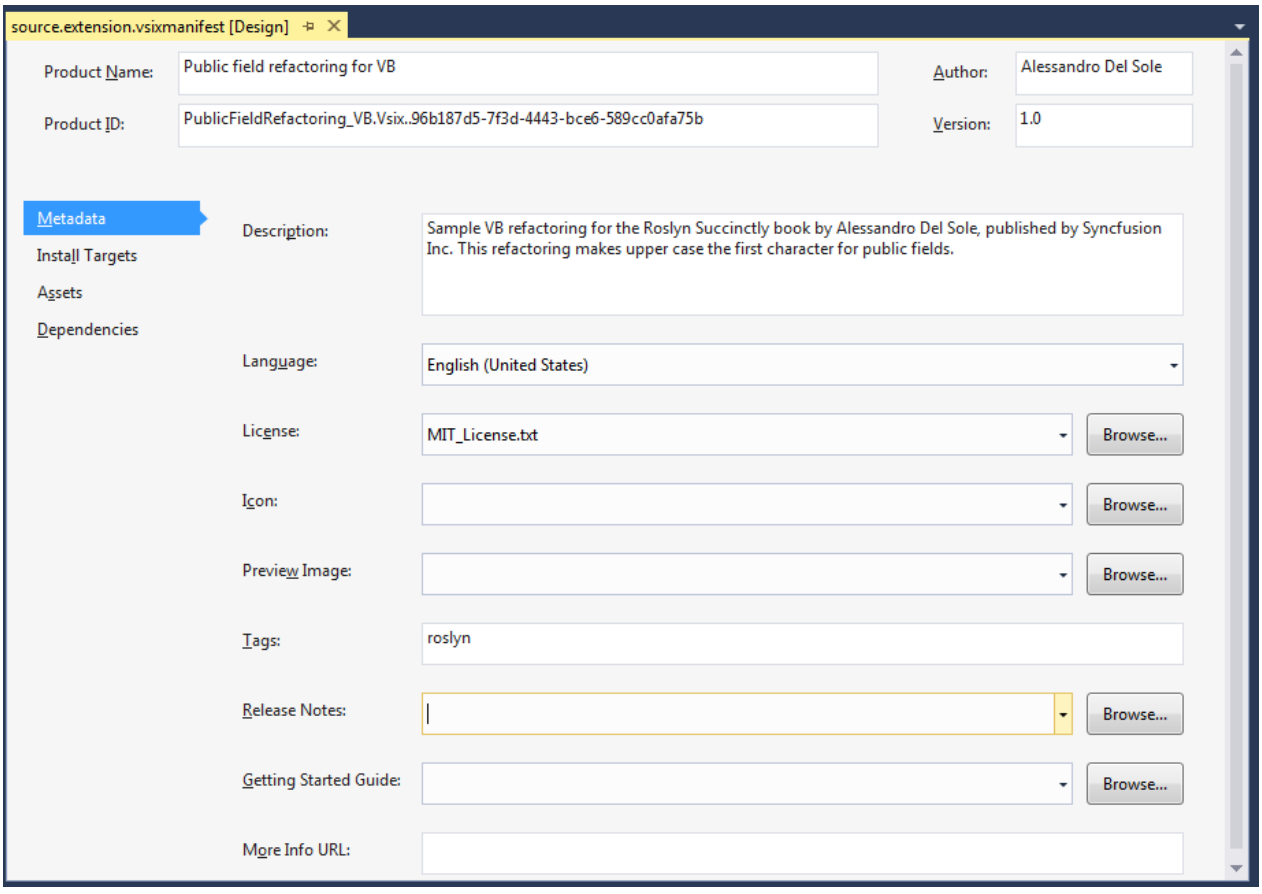

Figure 43: Editing the package manifest

The designer is composed of several tabs: Metadata, Install Targets, Assets, and Dependencies. In the Metadata tab, you supply all the information that will be shown to other developers in the Visual Studio Gallery and in the Extensions Manager tool, which is very important to identify and describe the package. One thing you should not change is the Product ID field, which contains a unique identifier containing a GUID for your product. You can edit the author name, the product name, and provide a full description of what the package does. You are also strongly encouraged to choose a license agreement, which must be a .txt file. The other information is optional.

Another important tab is Install Targets, where you specify the Visual Studio editions that can receive your package. When you open this tab, you will see a list of supported editions. At the time of writing, the Express editions are not supported, so select each Express edition, and then click **Delete**. You might want to include the Visual Studio 2015 Community edition instead, so click **New**. In the **Add New Installation Target** dialog box (see Figure 44), select the **Microsoft.VisualStudio.Community** option from the **Product Identifier** combo box. The Version Range field will be filled automatically. When you're done, click **OK**.

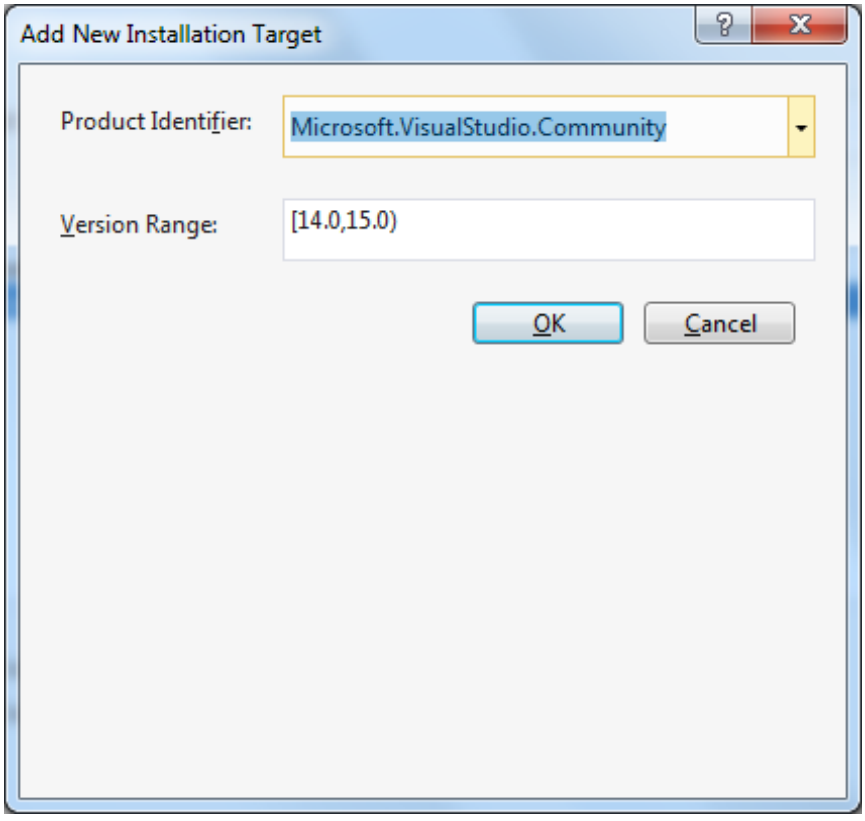

Figure 44: Specifying a supported edition of Visual Studio

At this point, the designer will show the newly added edition, plus the Visual Studio Professional edition supported by default, as demonstrated in Figure 45.

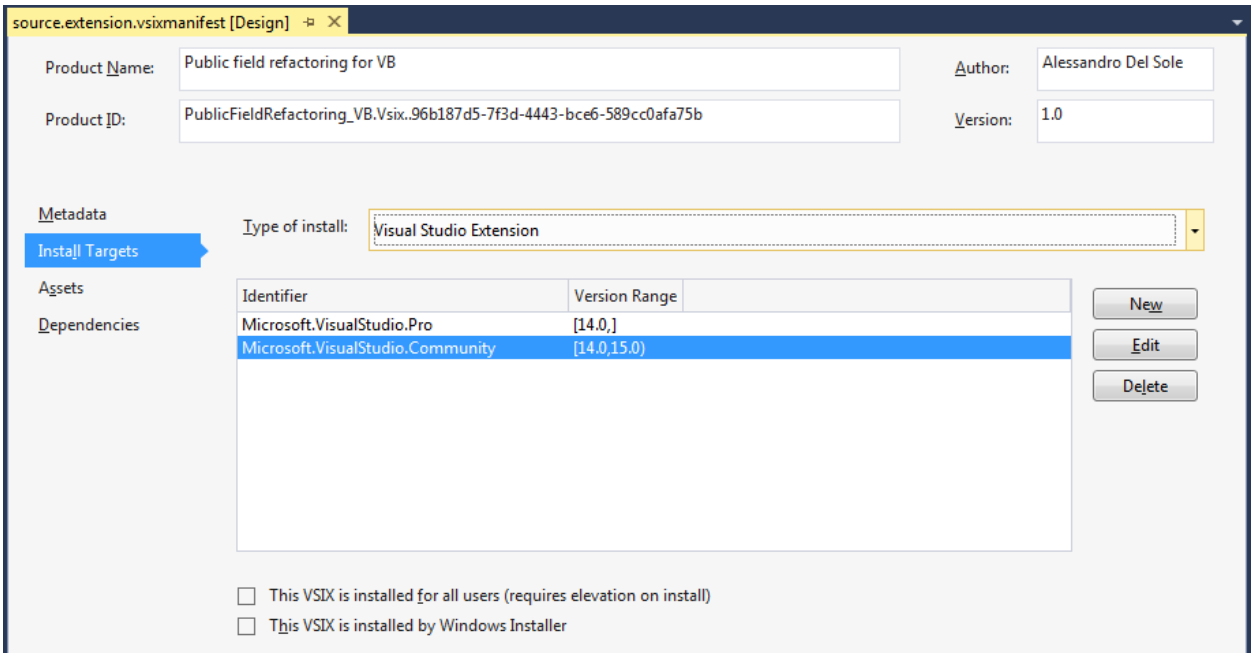

Figure 45: Supported editions of Visual Studio

Note: Pro edition means Professional and higher, so there is no need to specify the Enterprise edition, which is supported implicitly.

The Assets tab allows specifying additional project targets that will be packaged into the VSIX file. The Dependencies tab contains the list of frameworks and libraries that an extension depends on. In the case of analyzers and code refactorings, you do not need to edit the content of these tabs because they are already tailored for these types of libraries. However, it's worth mentioning that the Dependencies tab is where you can optionally specify any possible analyzer or refactoring dependencies. This is different from the NuGet experience, where dependencies are resolved automatically.

Once you have supplied the necessary information, ensure the build configuration is set to **Release** and compile the project. The build process produces a stand-alone, self-installing VSIX package that can be shared in a number of ways, or installed on the local machine by double-clicking. In the current scenario, the generated VSIX package will be published to the Visual Studio Gallery.

Tip: Changes you make in the package manifest are also visible in the experimental instance of Visual Studio when you debug a refactoring or an analyzer. To demonstrate this, start debugging with F5, and then open the Extensions Manager tool. If you select the current refactoring or analyzer, you will see the updated product information.

Publishing to the Visual Studio Gallery

Open the Visual Studio Gallery and sign in with your Microsoft Account using the **Sign In** link in the upper-right corner. When signed in, you will have options to upload new extensions and manage your published extensions (if any). Click **Upload** (see Figure 46) to start the Upload wizard.

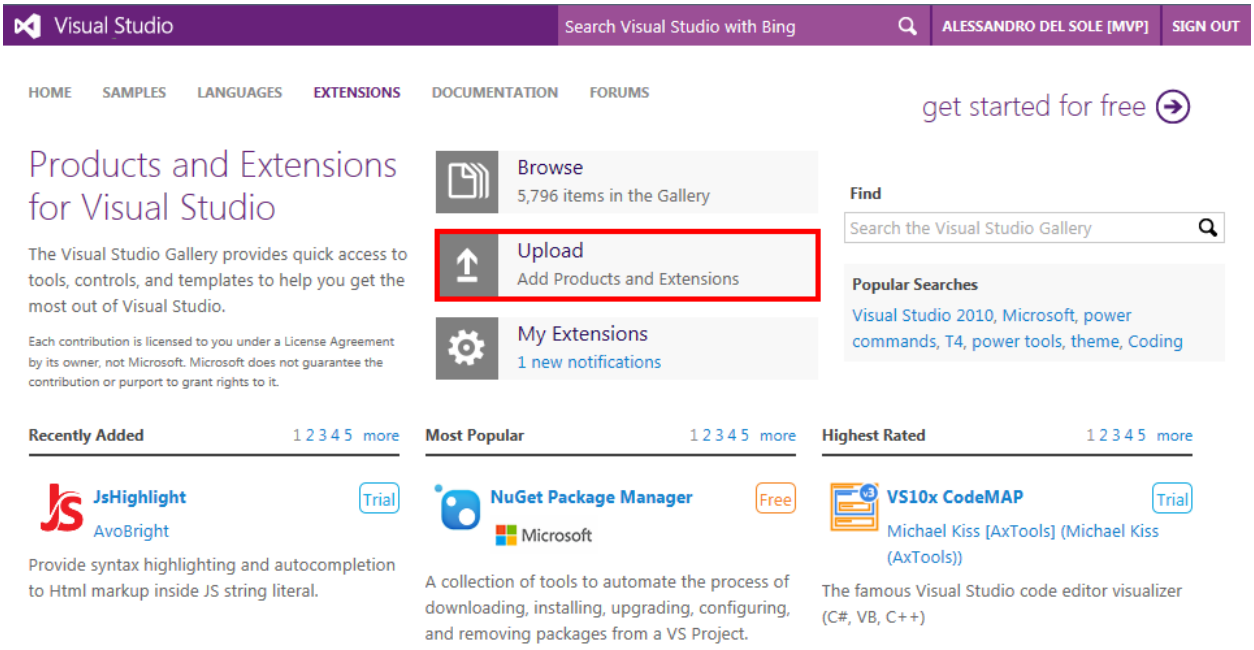

Figure 46: Visual Studio Gallery home page

The first thing you will do is specify the extension type. Select **Tool** (see Figure 47) and then click **Next**.

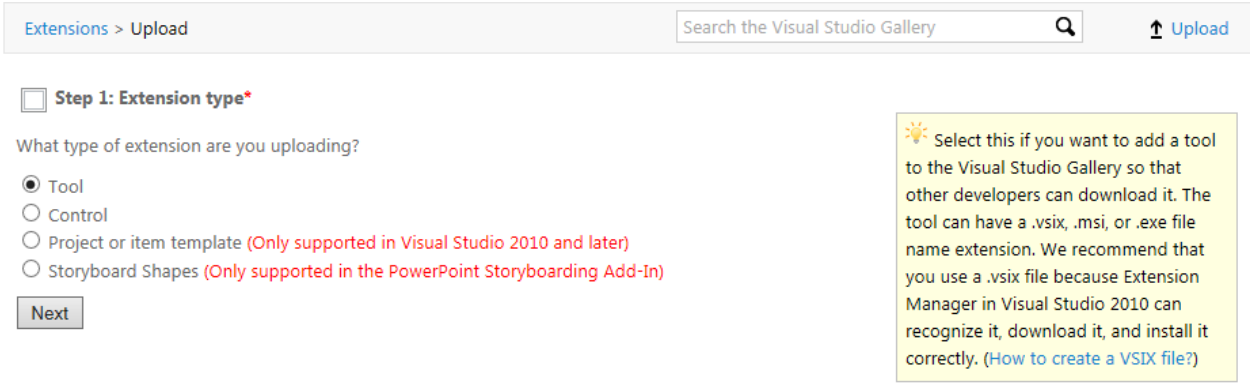

Figure 47: Specifying Tool as the extension type

In this step, you will specify the VSIX package to upload. Select **I would like to upload my tool**, and then click **Browse** (see Figure 48).

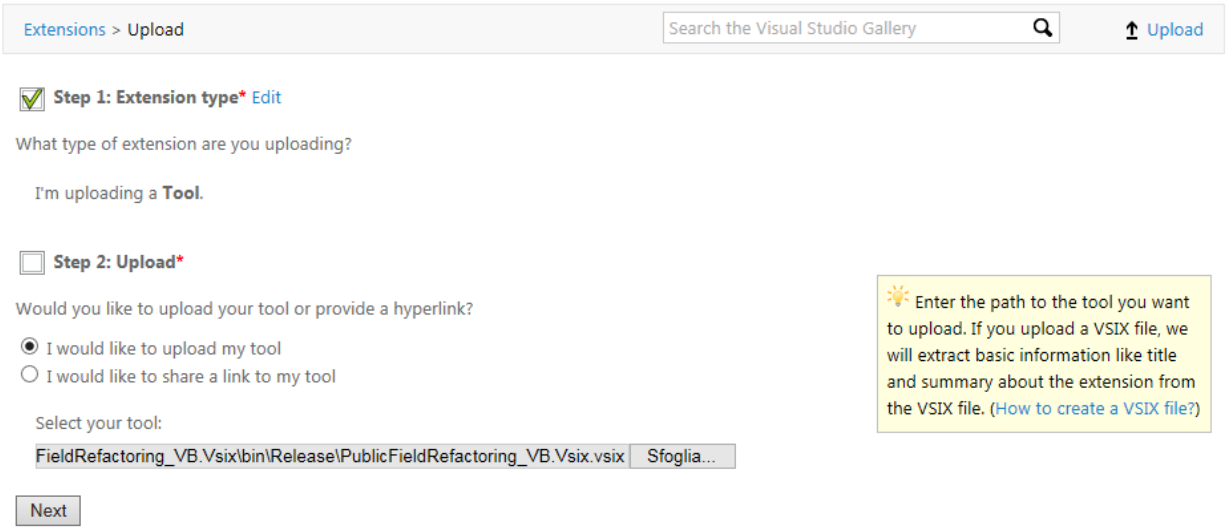

Figure 48: Specifying the VSIX package to upload

At this point, locate and select the VSIX package that contains the code refactoring. As a general rule, it is located under the **Bin\Release** subfolder of the VSIX project. In this case, it is **PublicFieldRefactoring_VB.Vsix\Bin\Release** for Visual Basic or **PublicFieldRefactoring_CS.Vsix\Bin\Release** for C#. Click **Next** when you're ready. In the next step (see Figure 49), you will see summary information that the Gallery takes from the package manifest, and you will be asked to do the following:

- Select up to three categories to help developers recognize the extension's purpose.
- Specify a cost category of Free, Trial, or Paid.
- Specify one or more tags that will help developers find your extension on the Gallery.

Categories and tags are totally your choice, but I recommend selecting **Coding** and **Programming Languages** as categories, and specifying **Roslyn** as a tag. In the **Cost category** drop-down menu, select **Free**. If you decide to use Trial or Paid, implementing the purchase logic is your own responsibility; the Visual Studio Gallery will basically inform users that an extension is not free.

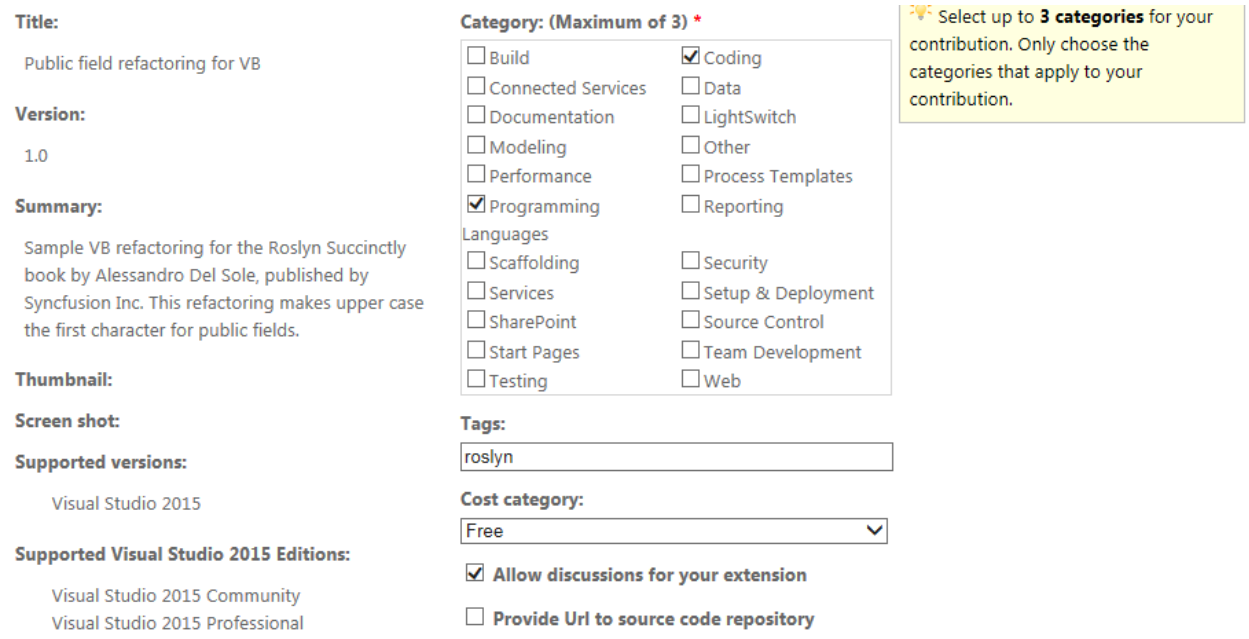

Figure 49: Categorizing your package

A detailed description of the package content must also be supplied. Scroll down the page and provide a description, as shown in Figure 50. You can take advantage of the built-in editor that also allows formatting text and adding multimedia.

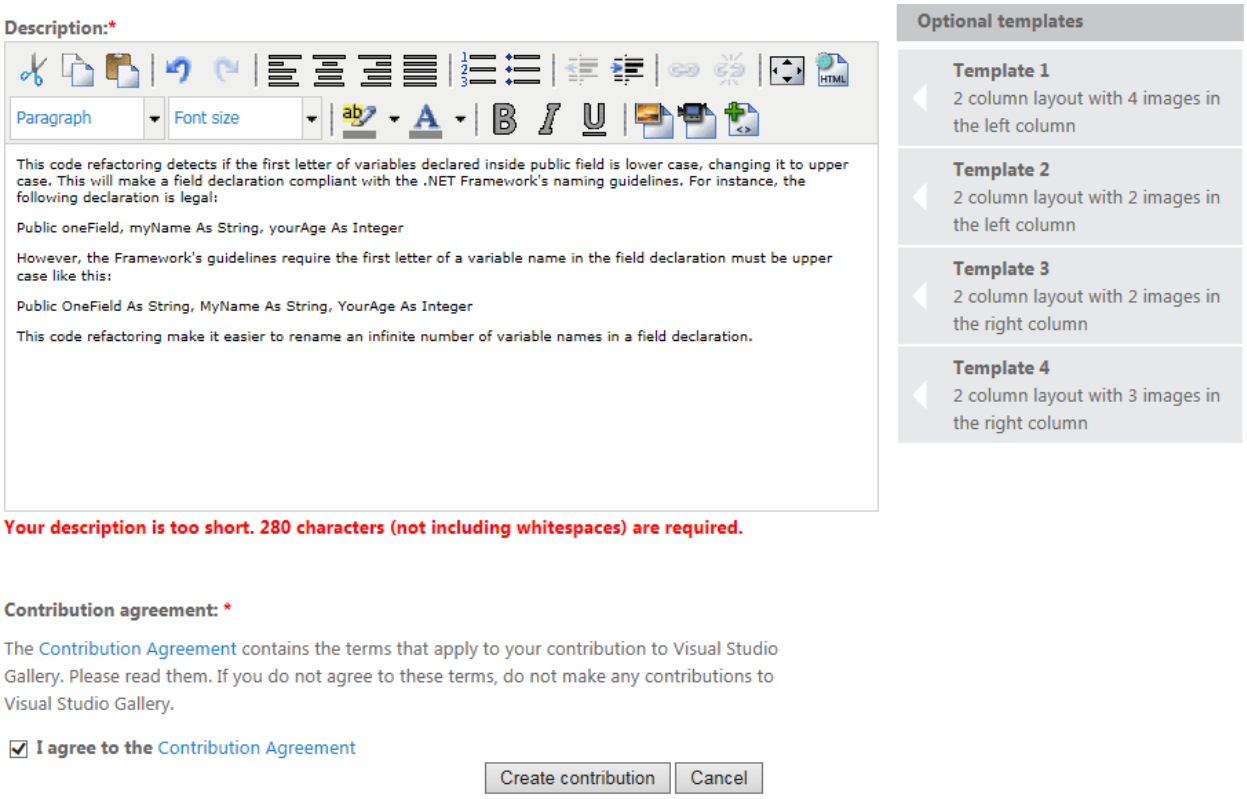

Figure 50: Providing the package description

Ensure your description is at least 280 characters long; otherwise, an error message will appear (see Figure 50). When you're done, read and accept the contribution agreement, and click **Create contribution**. At this point, your contribution has been created and you have an option to see a preview of how it will appear to other developers (see Figure 51).

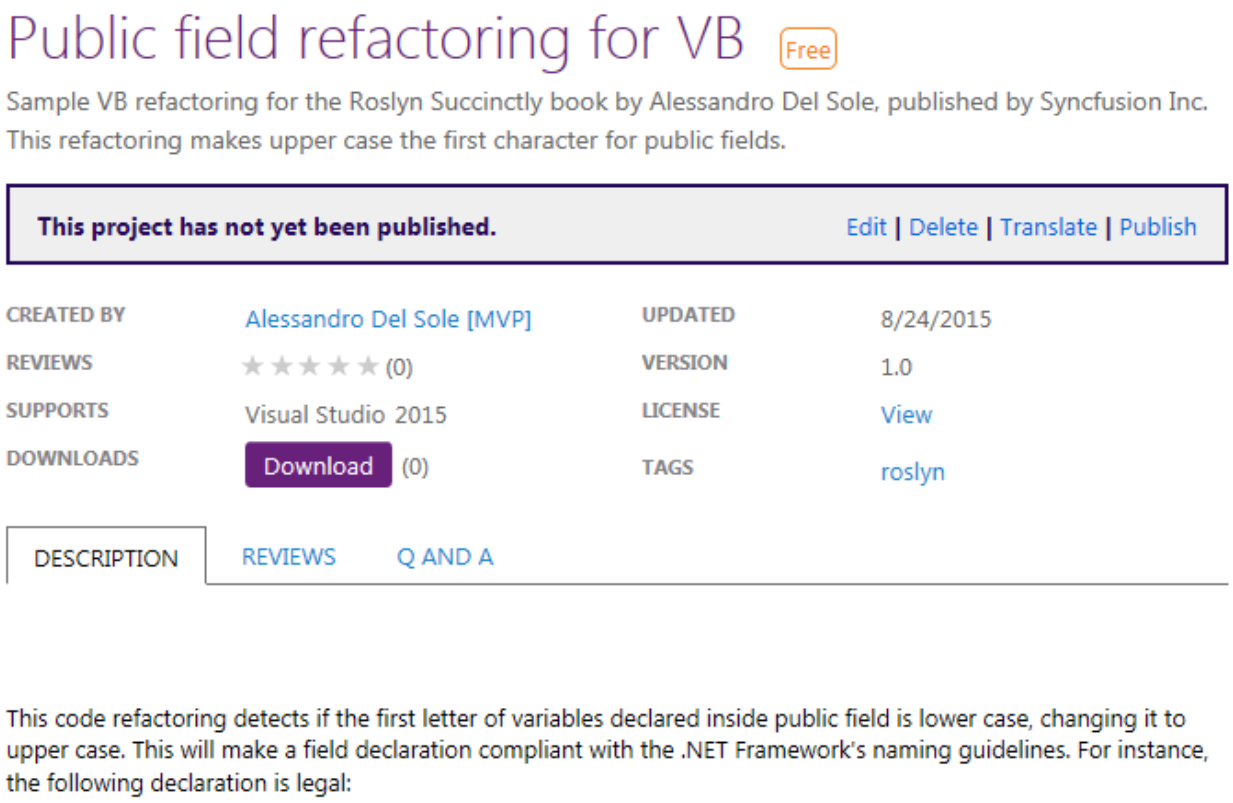

Figure 51: Reviewing your contribution before it's public

If you are satisfied with the preview and it matches your requirements, click **Publish** so that it will be immediately available for downloading. If you wish to make any modifications, click **Edit** instead. With a few simple steps, you have published a code refactoring to the Visual Studio Gallery. Remember that the same steps apply to analyzers, too.

Downloading and Installing Code Analyzers and Refactorings from the Visual Studio Gallery

To understand the developer experience with an analyzer or refactoring available from the Visual Studio Gallery, launch Visual Studio 2015, and then select **Tools** > **Extensions and Updates**. Select the **Online** tab, and in the search box, type **Roslyn**. As you can see in Figure 52, the code refactoring appears in the list of available extensions, also showing the package description.

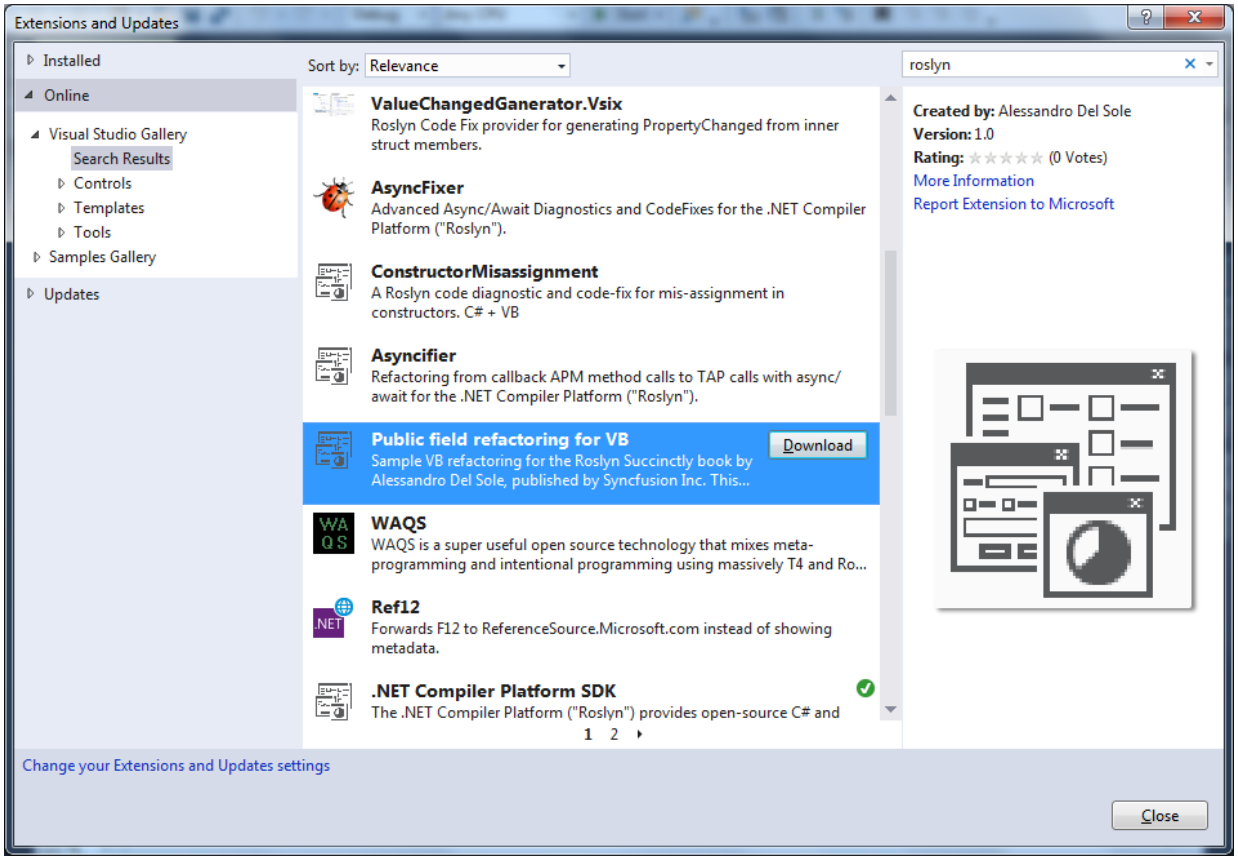

Figure 52: Finding a code refactoring in the Visual Studio Gallery

Click **Download**. When the download completes, you will be asked to accept the included license agreement (see Figure 53).

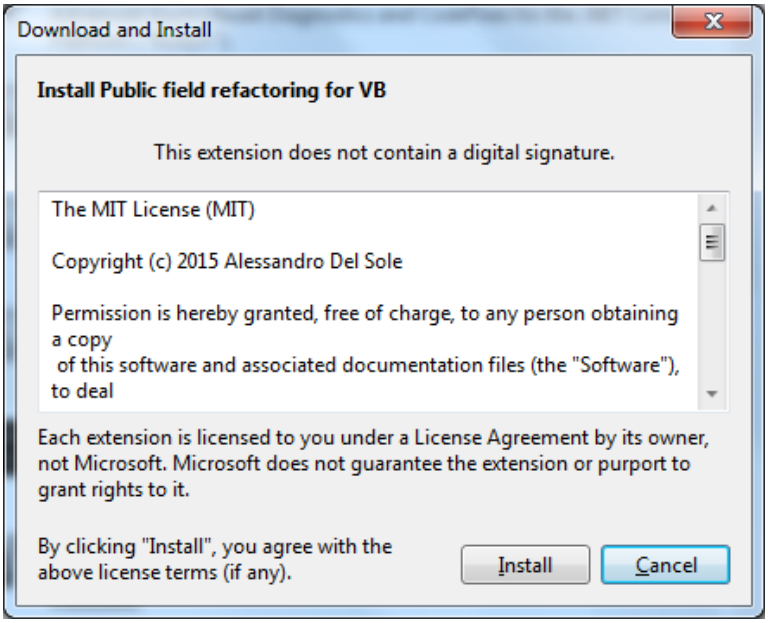

Figure 53: Accepting the package license agreement

When you click **Install**, Visual Studio 2015 will complete the install operation in a few seconds. Notice that you will be asked to restart Visual Studio after the installation completes. Once Visual Studio has restarted, create a new console application. Add a new class and a public field declaration in it, as shown in Figure 54.

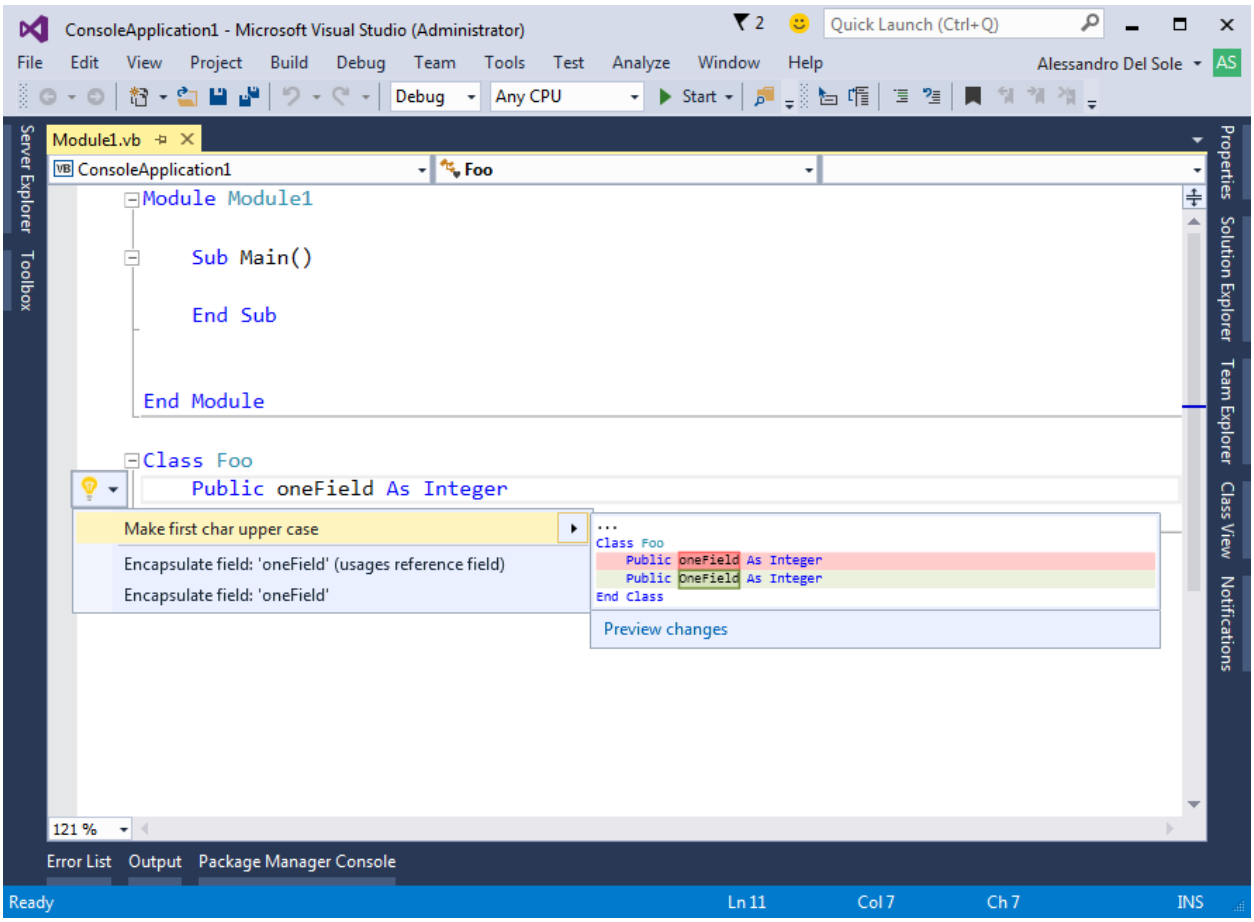

Figure 54: Testing the code refactoring

Right-click the **`Public`** keyword, select **Quick Actions**, and see how the newly installed code refactoring works perfectly, allowing you to change public field member identifiers to uppercase.

Uninstalling Code Analyzers and Refactorings

Analyzers and refactorings installed from the Visual Studio Gallery can be easily disabled or uninstalled. To accomplish this, select **Tools** > **Extensions and Updates**, and in the list of installed extensions, select the one you want to disable or remove. Figure 55 shows the code refactoring described in this chapter.

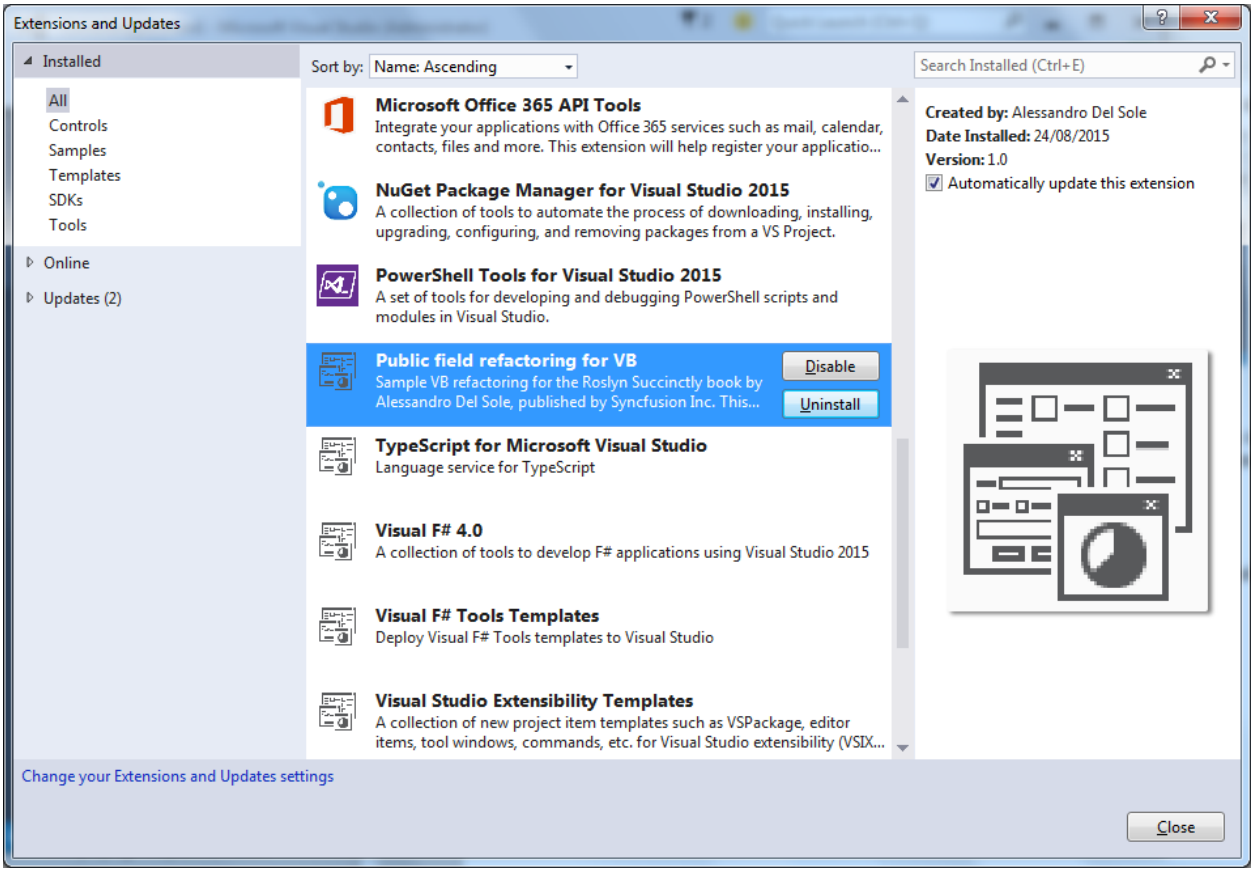

Figure 55: Disabling or uninstalling a code refactoring

Select **Disable** or **Uninstall**, but remember that the selected refactoring (or analyzer) will not be available to any projects until you explicitly re-enable or reinstall.

Differences between NuGet and the Visual Studio Gallery

There are a number of differences between sharing analyzers through NuGet and the Visual Studio Gallery. The following is a list of important differences that you might want to consider when you have to decide where to publish your work:

- NuGet packages are per project. Every time you create a new project, you will need to install the analyzer or refactoring from NuGet.
- VSIX packages are Visual Studio extensions, so they are always available to new and existing projects. You need to disable or uninstall packages explicitly if you do not want your projects to use them.
- NuGet packages automatically resolve any dependencies an analyzer or refactoring might rely on.
- VSIX packages do not automatically resolve any dependencies an analyzer or refactoring might rely on. You are allowed to add dependencies manually in the Dependencies tab of the package manifest.
- VSIX packages are self-installing. This means that you can share a .vsix file not only through the Visual Studio Gallery, but in any way you want. For example, you could

share one with a shared network path, email, or a USB flash drive. The recipient will simply double-click the file to install the extension. Installation is powered by the VSIX Installer engine, which integrates with the operating system.

- Updating packages with a new version is pretty similar. For NuGet packages, Visual Studio 2015 automatically updates the version number. For VSIX packages, you update the version number manually in the package manifest editor. For both, you upload the updated package in the respective website.

Of course, you are totally free to publish analyzers and refactorings to both NuGet and the Visual Studio Gallery.

Chapter Summary

Analyzers and code refactorings can be published to the Visual Studio Gallery in the form of VSIX packages. Project templates for both analyzers and refactorings automate the generation of a stand-alone VSIX package that can be shared with other developers in many ways. Visual Studio 2015 includes a convenient designer that you use to edit the package manifest, whose information will be shown to other developers. Concerning publishing, the Visual Studio Gallery allows publishing a VSIX package with a few steps, providing an option to review the package information before it is available to the public. Using analyzers and refactorings deployed in this way is very simple: the Extensions and Updates dialog in Visual Studio 2015 allows searching for and downloading both components in the same way you would search for and download any other Visual Studio extension.

Chapter 8 Workspaces, Code Generation, Emit

The .NET Compiler Platform is not just about analyzers and refactorings. In Chapter 3, you were introduced to the Workspaces APIs and the compiler pipeline, which includes the emit phase. This chapter provides a short introduction to the Workspaces APIs, and how you can implement code generation and emit an assembly.

Getting Started with Workspaces

The Workspaces APIs allow you to interact with everything that makes up an MSBuild solution, including projects, code files, and metadata, with objects exposed by the .NET Compiler Platform. Most of the necessary types are defined inside the Microsoft.CodeAnalysis.Workspaces.dll assembly, plus libraries that are tailored for a specific language, such as Microsoft.CodeAnalysis.CSharp.Workspaces.dll and Microsoft.CodeAnalysis.VisualBasic.Workspaces.dll.

In this chapter, you'll create a sample project that opens a solution (.sln file) and lists all the projects in that solution, plus all the code files (documents) and references of each project. For an easier starting point, you can take advantage of a project template called **Stand-Alone Code Analysis Tool**, which generates a console application with all the necessary references to the Roslyn APIs. Of course, you can manually add a reference to the Microsoft.CodeAnalysis.dll library in any project. This project template is located in the **Extensibility** node under the language of your choice, as shown in Figure 56.

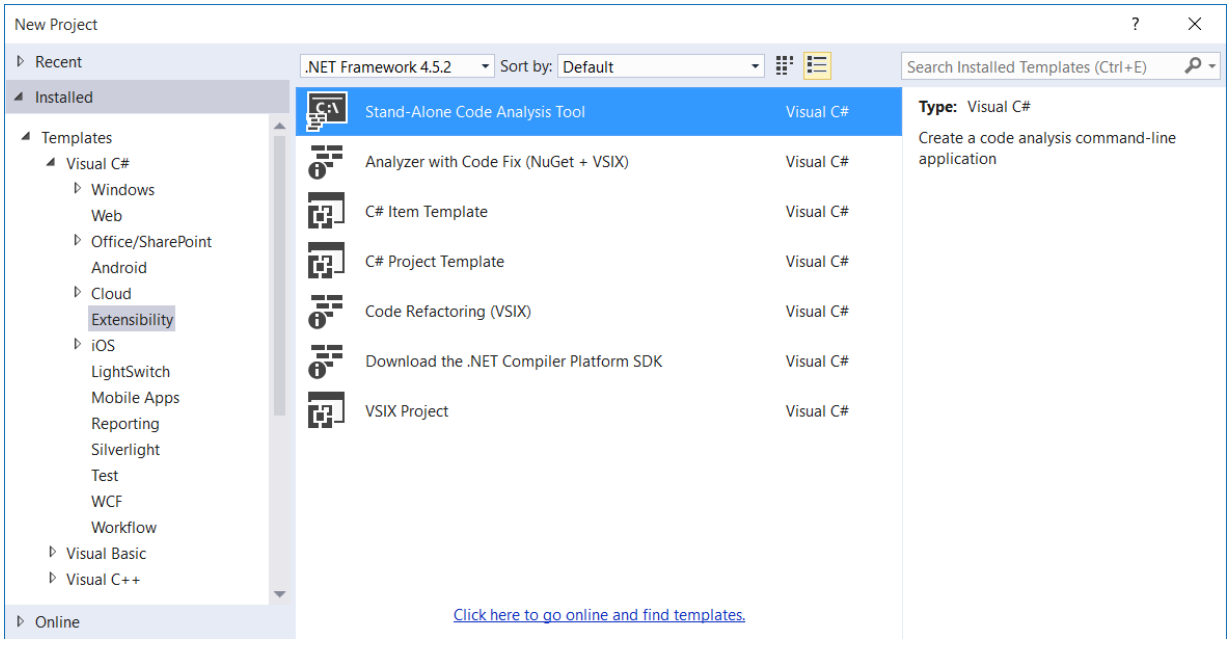

Figure 56: Creating a stand-alone code analysis application

To interact with a solution, you need to create an instance of the **`Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace`** class, which represents a workspace that can be populated with MSBuild solutions and projects. This class exposes a method called **`Create`**, which generates the workspace, and a number of methods to interact with solutions and projects, such as **`OpenSolutionAsync`** and **`OpenProjectAsync`**.

Tip: Using IntelliSense to discover the full list of available methods is almost always the best approach. The method names are self-explanatory, so understanding the purpose of each one will not be difficult.

Code Listing 19 demonstrates how to open an existing solution on disk, but remember to change the solution path to one existing on your machine.

Code Listing 19 (C#)

```
        static void Main(string[] args)
        {
            // Path to an existing solution
            string solutionPath =
                   "C:\\temp\\RoslynSolution\\RoslynSolution.sln";

            // Create a workspace
            var ws = Microsoft.CodeAnalysis.
                MSBuild.MSBuildWorkspace.
                Create();

            // Open a solution
            var solution =
                ws.OpenSolutionAsync(solutionPath).Result;

            // Invoke code to iterate items
            // in the solution
            // using a program-defined method
            IterateSolution(solution,
                solutionPath);
        }
```

Code Listing 19 (VB)

```
    Sub Main()
        'Path to an existing solution
        Dim solutionPath =
            "C:\temp\RoslynSolution\RoslynSolution.sln"

        'Create a workspace
        Dim ws =
            MSBuild.
```

```
            MSBuildWorkspace.Create()

        'Open a solution
        Dim solution =
            ws.OpenSolutionAsync(
            solutionPath).Result

        'Invoke code to iterate
        'items in the solution
        'using a program-defined method
        IterateSolution(
            solution, solutionPath)
    End Sub
```

The invocation to **`OpenSolutionAsync`** returns an object of type **`Solution`**, which represents an MSBuild solution. Similarly, invoking **`OpenProjectAsync`** would result in an object of type **`Project`**. **`Solution`** exposes a property called **`Result`**, of the same type, which returns the actual instance of the solution. The next step is iterating the list of projects in the solution and the list of documents and references for each project. To understand what the next piece of code will do, first take a look at Figure 57, which shows the result based on a sample C# WPF application.

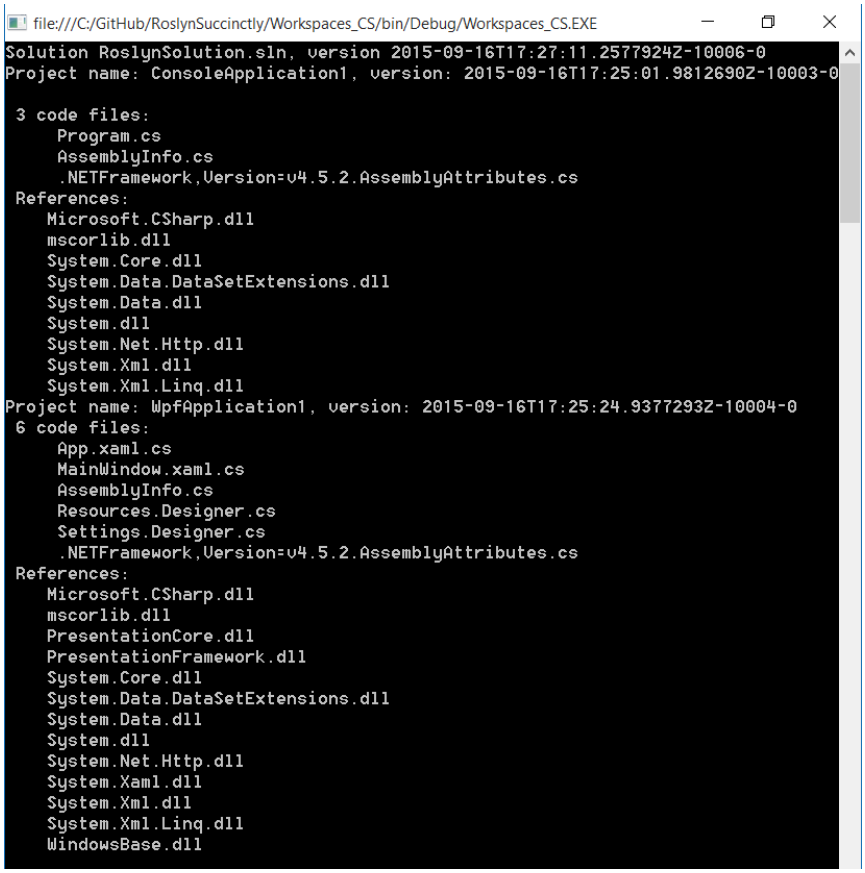

Figure 57: Iterating properties of an MSBuild solution

As you can see, you need some code that displays information about the solution, the list of projects, and the code files and references of each project. The code required to perform this is shown in Code Listing 20.

Code Listing 20 (C#)

```
static void IterateSolution(Solution solution,
             string solutionPath)
 {
     // Print solution's path and version
     Console.WriteLine(
         $"Solution {System.IO.Path.GetFileName(solutionPath)},
           version {solution.Version.ToString()}");

     // For each project...
     foreach (var prj in
              solution.Projects)
     {
         // Print the name and version
         Console.WriteLine(
             $"Project name: {prj.Name}, version:
             {prj.Version.ToString()}");
         // Then print the number of code files
         Console.WriteLine(
             $" {prj.Documents.Count()} code files:");

         // For each code file, print the file name
         foreach (var codeFile in
                  prj.Documents)
         {
             Console.
             WriteLine($"     {codeFile.Name}");
         }

         Console.WriteLine(" References:");

         // For each reference in the project
         // Print the name
         foreach (var reference in
                  prj.MetadataReferences)
         {
             Console.WriteLine(
          $"    {System.IO.Path.GetFileName(reference.Display)}");
         }
     }

     Console.ReadLine();
 }
```

Code Listing 20 (VB)

```
Private Sub IterateSolution(
            solution As Solution,
            solutionPath As String)

    'Print solution's path and version
    Console.WriteLine(
        $"Solution {IO.Path.
        GetFileName(solutionPath)},
        version {solution.Version.ToString}")

    'For each project...
    For Each prj In solution.Projects

        'Print the name and version
        Console.
        WriteLine(
        $"Project name: {prj.Name}, version: {prj.Version.ToString}")
        'Then print the number of code files
        Console.
        WriteLine($" {prj.Documents.Count} code files:")

        'For each code file, print the file name
        For Each codeFile In
            prj.Documents
            Console.
            WriteLine($"     {codeFile.Name}")
        Next

        Console.
        WriteLine(" References:")

        'For each reference in the project
        'Print the name
        For Each ref In
            prj.MetadataReferences
            Console.
            WriteLine($"    {IO.Path.GetFileName(ref.Display)}")
        Next
        Console.WriteLine("")
    Next

    Console.ReadLine()
End Sub
```

Among others, the **`Solution`** class exposes the **`Version`** property, which represents the solution's file version number. It also exposes the **`Projects`** property, which is a collection of **`Project`** objects, each representing a project in the solution. The **`Project`** class exposes a

property called **Documents**, a collection of **Document** objects, each representing a code file in the project. **Project** also exposes the **MetadataReferences** property, which represents a collection of assembly references, including CLR libraries.

Each reference is represented by an object of type **MetadataReference**, whose **Display** property contains the assembly name. This example just shows the surface of the power of the Workspaces APIs, but it gives you an idea of how you can work with solutions, projects, and documents in a .NET way. You might also want to consider the **AdHocWorkspace** class, which also offers members to create projects, documents, and code files, and add them to a solution.

Code Generation and Emit

Note: This section requires you to have knowledge of reflection in .NET. For the code blocks that use reflection, only the Roslyn-related snippets will be discussed in detail.

With the Roslyn APIs, you can take source code, compile it, and emit an assembly. Source code can be pure-text parsed to a syntax tree or an existing syntax tree. Then you can use the **Compilation** class to perform an invocation to the compiler.

In this chapter, you will see how to generate a syntax tree from pure source text, and then how to use the **Compilation** class to invoke the compiler for emitting an assembly. More specifically, you will write source code that checks if a text file exists, printing its content to the Console window. You will also see how to interact with diagnostics by causing intentional errors.

Before you start, create a new console application based on the Stand-alone Code Analysis Tool project template (see Figure 56) in the language of your choice. Actually, **Compilation** is an abstract class and the base type for the **VisualBasicCompilation** and **CSharpCompilation** classes, which are tailored for VB and C#. As you know, the **Microsoft.CodeAnalysis** namespace exposes the **VisualBasic** and **CSharp** classes, which offer members that are tailored to a language's specifications. Both offer an option to represent a source document under the form of a syntax tree via the **VisualBasicSyntaxTree** and **CSharpSyntaxTree** objects. Both classes expose a method called **ParseText** that you invoke to generate a syntax tree from pure source text. Code Listing 21 shows how to parse source text and generate a syntax tree.

Code Listing 21 (C#)

```
        private static void GenerateCode()
        {
            SyntaxTree syntaxTree =
                CSharpSyntaxTree.ParseText(@"
    using System;
    using System.IO;

    namespace RoslynSuccinctly
    {
        public class Helper
```

```
        {
            public void PrintTextFromFile(string fileName)
            {
                if (File.Exists(fileName) == false)
                {
                    Console.WriteLine(""File does not exist"");
                    return;
                }

                using (StreamReader str = new StreamReader(fileName))
                {
                    Console.WriteLine(str.ReadToEnd());
                }
            }
        }
    }");
    // more code goes here...
} //end GenerateCode()
```

Code Listing 21 (VB)

```
    'Generate a syntax tree
    'from source text
    Private Sub GenerateCode()
        Dim tree = VisualBasicSyntaxTree.ParseText("
Imports System
Imports System.IO

Namespace RoslynSuccinctly
    Public Class Helper

        Public Sub PrintTextFromFile(fileName As String)
            If File.Exists(fileName) = False Then
                Console.WriteLine(""File does not exist"")
                Exit Sub
            End If

            Using str As New StreamReader(fileName)
                Console.WriteLine(str.ReadToEnd())
            End Using
        End Sub
    End Class
End Namespace")
```

The next step is preparing the assembly's metadata. This requires getting an assembly name, which is accomplished via the **`System.IO.Path.GetRandomFileName`** method for demonstration purposes, and creating a list of references that the assembly will rely on. The list of references must be an array of **`MetadataReference`** objects. **`MetadataReference`** exposes a

method called **CreateFromFile**. If you pass a type to this method, it will be able to determine the assembly files that contain the corresponding namespace definitions, and get a reference. In this case, the sample code uses objects from the **System** and **System.IO** namespaces, so to get a reference to the containing assemblies, you can pass **Object** and **File** to each **MetadataReference** in the array. This is demonstrated in Code Listing 22.

Code Listing 22 (C#)

```
            //Get a random file name for
            //the output assembly
            string outputAssemblyName =
                System.IO.Path.GetRandomFileName();

            //Add a list of references from assemblies
            //By a type name, get the assembly ref
            MetadataReference[] referenceList =
                new MetadataReference[]
                {
                    MetadataReference.
                    CreateFromFile(typeof(object).
                    Assembly.Location),
                    MetadataReference.
                    CreateFromFile(typeof(File).
                    Assembly.Location)
                };
```

Code Listing 22 (VB)

```
        'Get a random file name for
        'the output assembly
        Dim outputAssemblyName As String =
            Path.GetRandomFileName()

        'Add a list of references from assemblies
        'By a type name, get the assembly ref
        Dim referenceList As MetadataReference() =
            New MetadataReference() _
            {MetadataReference.
            CreateFromFile(GetType(Object).
            Assembly.Location),
            MetadataReference.
            CreateFromFile(GetType(File).
            Assembly.Location)}
```

The next step is creating a new compilation. You invoke the **Create** method to invoke the compiler, passing as arguments the new assembly name, an array of syntax trees, an array of references, and compilation options. This is demonstrated in Code Listing 23.

Code Listing 23 (C#)

```
            //Single invocation to the compiler
            //Create an assembly with the specified
            //syntax trees, references, and options
            CSharpCompilation compilation =
                CSharpCompilation.Create(
                outputAssemblyName,
                syntaxTrees: new[] { syntaxTree },
                references: referenceList,
                options: new CSharpCompilationOptions(
                    OutputKind.DynamicallyLinkedLibrary));
```

Code Listing 23 (VB)

```
        'Single invocation to the compiler
        'Create an assembly with the specified
        'syntax trees, references, and options
        Dim compilation As VisualBasicCompilation =
            VisualBasicCompilation.
            Create(outputAssemblyName,
                   syntaxTrees:=New SyntaxTree() {tree},
                   references:=referenceList,
                   options:=New VisualBasicCompilationOptions(
                   OutputKind.DynamicallyLinkedLibrary))
```

You can incrementally add new syntax trees or references via the **AddSyntaxTrees** and **AddReferences** methods. Notice how the **CSharpCompilationOptions** and **VisualBasicCompilationOptions** types allow specifying, among others, the output type for the assembly.

The next step is emitting the IL code, which is accomplished by invoking the **Emit** method over the **Compilation** instance. If the emit phase fails, the code will iterate a list of diagnostics and print their messages. The **EmitResult** type, which is the type returned by **Emit**, exposes an **ImmutableArray<Diagnostic>**, which contains a list of diagnostics that exist in the code. Code Listing 24 demonstrates this.

Code Listing 24 (C#)

```
            //Create a stream
            using (var ms = new MemoryStream())
            {
                //Emit the IL code into the stream
                EmitResult result = compilation.Emit(ms);

                //If emit fails,
                if (!result.Success)
                {
```

```
                //Query the list of diagnostics
                //in the source code
                IEnumerable<Diagnostic> diagnostics =
                    result.Diagnostics.Where(diagnostic =>
                    diagnostic.IsWarningAsError ||
                    diagnostic.Severity ==
                    DiagnosticSeverity.Error);

                //Write ID and message for each diagnostic
                foreach (Diagnostic diagnostic in
                    diagnostics)
                {
                    Console.Error.
                        WriteLine("{0}: {1}",
                        diagnostic.Id,
                        diagnostic.
                        GetMessage());
                }
            }
            else
            {
                //If emit succeeds, move to
                //the beginning of the assembly
                ms.Seek(0,
                    SeekOrigin.Begin);
                //Load the generated assembly
                //into memory
                Assembly inputAssembly =
                    Assembly.Load(ms.ToArray());

                //Get a reference to the type
                //defined in the syntax tree
                Type typeInstance =
                    inputAssembly.
                    GetType("RoslynSuccinctly.Helper");

                //Create an instance of the type
                object obj =
                    Activator.CreateInstance(typeInstance);

                //Invoke the method. Replace MyFile.txt with an
                //existing file name
                typeInstance.
                    InvokeMember("PrintTextFromFile",
                    BindingFlags.Default |
                    BindingFlags.InvokeMethod,
                    null,
                    obj,
                    new object[]
```

```
                    { "C:\\MyFile.txt" });
            }
        }
    } //end GenerateCode()
```

Code Listing 24 (VB)

```
    'Create a stream
    Using ms As New MemoryStream()
        'Emit the IL code into the
        'stream
        Dim result As EmitResult =
            compilation.Emit(ms)

        'If emit fails,
        If Not result.Success Then
            'Query the list of diagnostics in the source code
            Dim diagnostics As _
                IEnumerable(Of Diagnostic) =
                result.Diagnostics.Where(Function(diagnostic) _
                diagnostic.IsWarningAsError _
                OrElse diagnostic.Severity =
                DiagnosticSeverity.[Error])

            'Write ID and message for each diagnostic
            For Each diagnostic As _
                Diagnostic In diagnostics

                Console.Error.WriteLine("{0}: {1}",
                                        diagnostic.Id,
                                        diagnostic.GetMessage())
            Next
        Else
            'If emit succeeds, move to
            'the beginning of the assembly
            ms.Seek(0, SeekOrigin.Begin)

            'Load the generated assembly
            'into memory
            Dim inputAssembly As Assembly =
                Assembly.Load(ms.ToArray())

            'Get a reference to the type
            'defined in the syntax tree
            Dim typeInstance As Type =
                inputAssembly.
                GetType("RoslynSuccinctly.Helper")
```

```
            'Create an instance of the type
            Dim obj As Object =
                Activator.
                CreateInstance(typeInstance)

            'Invoke the method. Replace MyFile.txt with an existing
            'file name
            typeInstance.
                InvokeMember("PrintTextFromFile",
                             BindingFlags.Default Or
                             BindingFlags.InvokeMethod,
                             Nothing, obj,
                             New Object() _
                             {"C:\MyFile.txt"})
        End If
    End Using
End Sub
```

To test the code, invoke the **GenerateCode** method in the **Main** method. If you supply the name of an existing text file, you will see how its content will be printed to the Console window. This is possible because the source text you wrote is parsed into a syntax tree and compiled into a .dll assembly. Other than testing how the code properly works against an existing text file, you can try causing intentional errors in the source text and then run the code. Because the compiler's job is not only generating IL code but also performing code analysis, at that point it will report the proper diagnostics; these can be analyzed and utilized as required. For example, you can change **Writeline** to **Witeline** and **StreamReader** to **SreamReader**. Figure 58 shows the result of the code analysis.

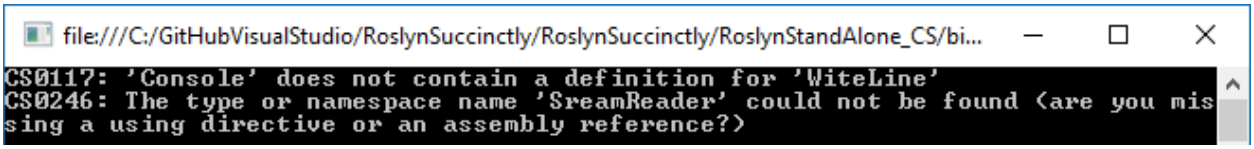

Figure 58: The result of code analysis over Roslyn-generated source code

In summary, the Roslyn APIs are very powerful and useful, not only to generate and compile code, but also for performing code analysis over the source code.

Chapter Summary

The .NET Compiler Platform is not just analyzers and code refactorings. In this chapter, you were given an overview of the Workspaces APIs and the Emit APIs. You first saw how to interact with MSBuild solutions by using workspaces and the **Solution**, **Project**, and **Document** objects. In the second part, you saw how you can leverage the **Compilation** class to generate an assembly based on a parsed syntax tree, how you can invoke types from the resulting library, and how you can investigate diagnostics in the source code.